Efficient simulation of challenging PDE problems on CPU and GPU clusters

Von der Fakultät für Mathematik und Physik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Malte Schirwon

aus Köln

Hauptberichter: Prof. Dr. Dominik Göddeke

Mitberichter: Prof. Dr. Holger Steeb
Prof. Dr. Stefan Turek

Tag der mündlichen Prüfung: 28.04.2021

Institut für Angewandte Analysis und Numerische Simulation
der Universität Stuttgart
2021

Acknowledgments

I would like to express my sincere gratitude to my advisor Professor Dominik Göddeke for his support, guidance and patience during my time at the University of Stuttgart. He has accompanied and influenced my scientific career since since my Bachelor's studies and is largely responsible for my exploring the exciting field of high performance computing and hardware oriented numerics. I also want to express my gratitude for the many interesting and instructive collaborations that resulted from his connections.

More acknowledgments go to my dear colleagues who come from the other disciplines of applications considered in this work.

The first cooperation partner I would like to thank is Marius Brehler, with whom I had my first scientific cooperation, which was completed with several papers and conference visits. Chapter 3 of this thesis could not have been written without him. Furthermore, I would like to thank Nadine Kijanski, David Krach, Maria Osorno and Rakulan Sivanesapillai from the field of continuum mechanics, with whom our SPH toolkit for image-based flow simulations on the pore scale of porous media was developed. In this context, I would like to thank Prof. Holger Steeb, the advisor of the people mentioned above, and who led this project together with Prof. Dominik Göddeke. In the course of this collaboration, many results beyond the ones described in chapter 4 have been obtained.

I have to thank Dimitri Komatisch for another insight into a subject area that was almost unknown to me before. Since he sadly passed away and the thoughts hurt, it is difficult for me to choose the right words. I am very appreciative to him for much conversation, discussion and ideas on the subject of seismic waveform modeling and inversion. In particular, I would like to thank him for allowing me to spend three months with him on site in Marseille during our collaboration. In the context of this collaboration I also got to know Vadim Monteiller, with whom my work in the field of seismic waveform modeling and inversion was further intensified. Vadim was available for any questions and could always help me with his expertise. I am glad that this collaboration will continue beyond this work.

Also, I would like to thank Professors Holger Steeb and Stefan Turek for their willingness to be the co-examiners of this thesis.

Besides these people who are directly related to this work, there are of course others I have to thank. I would like to thank my colleagues from IANS and especially the colleagues from my group from CMCS for the pleasant working atmosphere and the helpful discussions on any numerical topics. In particular, I would like to mention Mirco Altenbernd, Alexander Grimm, Felix Huber, Aaron Krämer, Julia Kühnert and Anna Rörich. During my time in Stuttgart I shared an office with Mirco and not only because of that he was always my first contact person for any numerical or mathematical questions.

Felix already supported me as a student worker from 2016 onwards, so that I was able to draw on his expertise at an early stage. Therefore, I would like to thank him for his contributions that were made in the context of this research and that are partly included in chapter 4 of this thesis.

I would like to conclude my words of acknowledgement with thanking all those whom I cannot mention by name here, as there would be too many. Many thanks to my friends and family who have supported me during this time.

During my time as doctoral student, I have been partly funded by the German Research Foundation (DFG) as part of the Cluster of Excellence in Simulation Technology (EXC 310) at the University of Stuttgart, and it is gratefully acknowledged.

Contents

Zusammenfassung

Der wichtigste Beitrag dieser Dissertation ist zu zeigen, wie effiziente Parallelisierungstechniken für numerische Simulationen von partiellen Differentialgleichungen (PDEs) entwickelt werden können und worauf zu achten ist, um eine möglichst gute Performance zu erhalten. Dazu reichen die Zielplattformen von leistungsstarken Workstations über kleine Cluster bis hin zu Supercomputern, wobei wir im speziellen Plattformen betrachten, die mittels Grafikkarten beschleunigt sind. Wir betonen, dass die effiziente numerische Simulation von PDE-Problemen auf neuartige Weise Aspekte aus der numerischen Analyse, den numerischen Methoden (Algorithmen, Datenstrukturen und andere Bereiche, die eher der Informatik zuzuordnen sind) und Hardware-Details umfasst und kombiniert.

Unzählige Modelle in Naturwissenschaft, Technik und Ökonomie basieren auf Systemen von PDEs. Die Wahl der Modellierungsverfahren, die Implementierung numerischer Lösertechniken sowie die gewählte Zielplattform beschränken die Genauigkeit und Dauer der Simulation. Eine Erhöhung der Genauigkeit und/oder Reduktion der Dauer der Simulation ist in der Regel nicht ohne effiziente Software möglich. Anhand drei Anwendungsszenarien, die zum Teil auf unstrukturierten Daten und Strukturen basieren, passen wir bereits existierende Methodiken und Algorithmen auf die Zielplattformen an oder verändern die Implementierungsweise, um so eine optimale Effizienz zu erreichen.

Beim ersten Anwendungsfall handelt es sich um die Wellenausbreitung in Lichtwellenleitern. Wir stellen eine MPI-parallele Implementierung vor, die insbesondere für kleine Cluster geeignet ist. Der zweite Anwendungsfall ist der Fluss in porösen Medien. Anhand dieser beiden Anwendungungen entwickeln wir Implementierungstechniken, welche deren Effizienz steigern. Ebenso stellen wir angepasste Version eines Nachbarschaftsalgorithmus vor, der für aktuelle Grafikkarten eine weitere Effizienzsteigerung erzielt.

Diese gesteigerte Effizienz und damit verringert Laufzeit erlaubt es, aufwendigere Simulationen zu betrachten. Eine solche Anwendung ist die dritte Anwendung, welche die Ausbreitung von seismischen Wellen und die Invertierung dergleichen darstellt. Die effizienten Implementierungstechniken, erlauben den Übergang von elastichen zu viskoelastischen Materialien bei die Invertierung von seismischen Wellen. Wir stellen ein Invertierungsschema vor, das es ermöglicht, auch die Dämpfungsparameter des viskoelastischen Materials zu invertieren. Weiter werden verschiedene Regularisierungsmethoden verglichen und ein modifiziertes, effizienteres Löserverfahren für solche Probleme vorgestellt.

Abstract

The main contribution of this dissertation is to show how efficient parallelization techniques for numerical simulations of partial differential equations (PDEs) can be developed and which aspects have to be considered in order to obtain the best possible performance. For this purpose, the target platforms range from high-performance workstations to small clusters and up to supercomputers. In particular, we focus on platforms accelerated by graphics cards. We emphasize that the efficient numerical simulation of PDE problems comprises and combines, in novel ways, aspects from numerical analysis, numerical methods (algorithmics, data structures and other areas more related to computer science) and hardware details.

Many models in science, engineering and economics are based on systems of PDEs. The choice of modeling techniques, the implementation of numerical solution techniques, as well as the chosen target platform limit the accuracy and the duration of the simulation. Increasing the accuracy and/or reducing the duration of the simulation is usually not possible without efficient software. Based on three application scenarios, we adapt already existing methodologies and algorithms to the target platforms or change the way they are implemented in order to achieve optimal efficiency. As a guiding scheme, we consider the challenging case of unstructured data and schemes.

The first application is the wave propagation in optical fibers. We present an MPI-parallel implementation that is particularly suitable for small clusters. The second application scenario is the flow in porous media. Based on both applications, we develop implementation techniques that increase their efficiency. Furthermore, we present an adapted version of a neighborhood algorithm that further increases the efficiency for current graphics cards.

The increased efficiency and reduced runtime allows to perform more complex simulations. One of theses applications is considered to be the third application, which is seismic wave propagation and waveform inversion. The feasibility of developing efficient implementations for computationally powerful target platforms permits us to consider the inversion of seismic waves in viscoelastic materials. In particular, we present an inversion scheme that also allows us to determine the damping parameters of the viscoelastic material. In addition, regularization methods and a modified solver method are presented, which can be used for a more efficient solution of such problems.

1 Introduction

1.1 General motivation

Since physical experiments are often too expensive or not feasible, they are replaced by numerical simulations. Partial differential equations (PDEs) can be used to describe a wide variety of phenomena like heat conduction, diffusion, sound electrostatics, electrodynamics, fluid dynamics, elasticity, gravitation, geodynamics, wave propagation and quantum mechanics. All these different physical phenomena can be expressed in a similar way in the form of (systems) PDEs, for which no closed-form solution exists.

In this way, numerical simulations are used, for example, to test new technologies. In the field of communications technology, new techniques are needed to cope with the increasing demand of data transfer. New techniques are also needed to increase the bandwidth in optical fibers. In order to test whether these techniques are applicable in practice or whether, for example, different signals interfere with each other and are therefore useless, numerical simulations can be used. Such a numerical simulation saves time and money, especially if the techniques turn out to be useless.

Another field where numerical simulations can replace laboratory experiments is in the analysis of properties of rocks. Such properties are needed, for example, to test whether a rock is suitable for a geothermal energy extraction process. Although a sample of this rock is still needed for the numerical simulation, which is digitized by means of a CT scan, the generated digital model can now be used for any number of simulations.

Numerical simulations can also be used to find such possible rock regions. Such imaging techniques are also used to find oil reservoirs in industry, or to evaluate computer tomographies in medicine. These methods are not used to save money on laboratory tests, but rather because these simulations obviously cannot be replaced by laboratory or other tests.

These three examples will reappear in chapters 3, 4 and 5 as we use these simulations to illustrate the development of efficient implementations, optimizations of numerical schemes, and the development of novel numerical methods.

There are two main aspects why designing and implementing efficient simulation soft-

ware for PDE problems is an ongoing and highly important research area.
The first aspect is that advancing science requires the development of next generation computational models to satisfy the accuracy and fidelity needs of targeted problems [51]. The potential impact of these models on computational science is twofold. To begin with, scientists will be able to account for more aspects of the physical phenomena being modeled. In addition, increases in the resolution of the system variables, such as the number of spatial zones, time steps, or particles, will improve simulation accuracy. Both of these impacts will place higher demands on computational hardware and software [51].

To meet these needs, vast amounts of computing power are often needed, reaching and pushing the limits of the fastest supercomputers. During the pursuit of this thesis, the peak performance of the fastest system increased from 54.9 PFLOP/s (Peta floating point operations per second) [152] to 537.2 PFLOP/s [153]. This means that the peak performance must roughly double again to reach the exascale threshold of 10^{18} FLOP/s. The increase in peak performance from TFLOP/s (i.e. 10^{12} FLOP/s) to PFLOP/s (i.e. 10^{15} FLOP/s) took 12 years from 1997 [149] to 2009 [150]. More recently the peak performance doubled with the release of the current fastest supercomputer Fugaku [151, 153]. Therefore it is expected that the exascale threshold could be reached in the next 1 to 2 years. Driven mostly by power constraints, exascale-class machines will see a massive increase in the number of computing units, whether homogeneous cores or heterogeneous mixtures of multipurpose CPUs and specialized processing units such as GPUs. Figure 1.1 shows that the number of NVIDIA GPU accelerators has increased significantly in recent years. For instance, more than 50% of the top 20 fastest supercomputers contain GPUs.

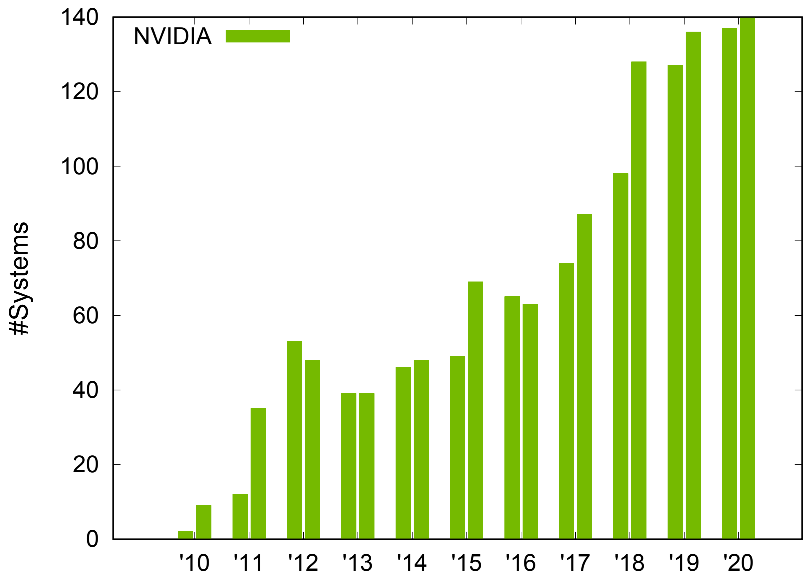


Figure 1.1: Number of systems accelerated by NVIDIA GPUs in the TOP500 list [153].

Furthermore, memory and bandwidth will not increase as quickly as core count, and data transfer latencies will be exposed further [106, 141, 155]. The anticipated exascale architectures will present significant challenges for scalable software development (see e.g.,

[71, 137]).
Nevertheless, efficient programming alone is not sufficient. At least as important is the scaling of the underling algorithms. Ideal strong scaling means that the compute time is halved if the amount of compute power is doubled, while the problem size remains constant. Ideal weak scaling means constant compute time when resources and problem size are simultaneously doubled. A more detailed definition of the scaling concept can be found in section 1.3. Therefore, the role of applied mathematics in the exascale effort is highly important and must be sufficiently investigated: In the past, numerical algorithms and libraries have contributed as much to increases in computational simulation capability as improvements in hardware. The expected developments in computer systems will place an even greater focus on numerical algorithms as a means of increasing the computational capability. Significant new model development, algorithm redesign, and science application code reimplementation will be required in order to effectively exploit the power of exascale architectures [51, 71, 137]. This thesis contributes to all of these aspects, with a focus on numerical algorithms.

The second is an economic and ecological aspect. It is first and foremost an economic aspect to obtain the simulation results as fast as possible, but without losing accuracy. Or it is necessary to get the results in real time or even faster than real time, e.g., collision protection in autonomous driving, or earthquake prediction respectively. Additionally there is also an ecological interest to use the hardware as efficiently as possible in order not to consume more power and therefore produce more carbon dioxide or excess heat than necessary. It also makes sense to use hardware that is energy efficient. Not only should it have a positive side effect to use less energy through energy efficient hardware, but it could also be a (small) step towards less CO_2 production. Therefore we develop software which runs on GPU accelerators. GPUs excel at energy efficiency, making them more and more likely candidates for future exascale systems. This becomes particular clear when comparing the TOP500 with the GREEN500 lists (November 2020)[64, 153], which provides a ranking of the most energy-efficient supercomputers in the world. GPUs are also often used as a target platform in the field of artificial intelligence and deep learning, since they are well suited for training neural networks with thousands to millions of input samples. As artificial intelligence and deep learning have gained and continue to gain importance in many areas of science in recent years, it is expected that the number of GPUs in small clusters, but also in large supercomputers, will increase. Also, it is worth noting that GPU computing does not only target big machines, even high-performance workstations contain one or more GPUs. Substantial speedups have been demonstrated for problems that fit into the memory of a single workstation, somewhat avoiding the need to move towards MPI-parallel implementations for strong scaling to keep runtime acceptable for increasing problem sizes and/or model complexities. On average, the theoretical peak performance of GPUs continues to level at roughly 2–5x above that of CPUs in terms of flop rates, and 3–7x for memory bandwidth. These numbers hold for the ‘fair’ socket-vs-socket comparison, and can translate to speedups in the same range for many applications [117, 141]. So we take both the ecological and the economic aspects into account.

The main focus of this thesis is to present different aspects, challenges, and solutions for implementing efficient software, to numerically solve challenging, representative PDE problems. In particular, we focus on GPU systems. Target platforms range from workstation-type systems equipped with (multiple) GPUs, small to medium-size GPU clusters both on-site and in the cloud, and GPU-accelerated supercomputers. In order to write efficient software, it is necessary to know the requirements of the software, the underlying target platform and its architecture, and to devise a scalable numerical algorithm.

1.2 Different types of hardware

We briefly sketch the main conceptual aspects and differences between the mentioned target platforms based on CPUs and GPUs, and also highlight several recent trends that could be interesting in future work.

Probably best known is the Central Processing Unit (CPU), which is the heart of a computer. The CPU executes basic arithmetic, logic, control, and input/output operations specified by instructions in the program. A multi-core CPU comprises two or more separate processing units called cores, each of which reads and executes program instructions as if the computer had multiple CPUs. In simple terms, a multi-core processor can be thought of several CPUs sitting on the same chip and sharing RAM (Random-Access Memory) and on-die memory.

The graphics processing unit (GPU) was first introduced 1999 to offload simple graphics operations from the CPU to a new, additional, dedicated processor [118]. As graphics expanded into 2D and, later, 3D rendering, GPUs became more powerful. Highly parallel on-chip operations are highly advantageous when processing an image composed of millions of pixels, so current-generation GPUs include thousands of 'cores' or processing units designed for efficient execution of mathematical functions. The Tesla V100, one of NVIDIA's latest devices introduced in 2017, contains 5,120 CUDA cores for single-cycle multiply-accumulate operations and 640 tensor cores for single-cycle matrix multiplication. GPUs have spread far beyond their initial application, because many algorithms in other fields lend themselves to parallel execution. Many of the world's fastest supercomputers, for example number 2 and 3 of the TOP500 (November 2020) [153], include thousands of both GPUs and CPUs. More differences between CPU and GPU are described in chapter 2.1.

Field-programmable Gate Arrays (FPGA) consist of internal hardware blocks with user-programmable connections to customize operations for a specific application [9]. In simple terms, they can be considered as a series of programmable blocks interconnected by programmable links. In contrast to the other processors mentioned, the connections between the blocks can be reprogrammed, changing the internal operation of the hardware. That means that its programmable structure can be configured to implement any combination of digital functions. Also, algorithms can be implemented in a massively parallel manner, which means that a huge amount of data processing is possible. This flexibility makes the FPGA the processor of choice for applications where standards are evolving, such as digital television, consumer electronics, cyber security systems and wireless communications.

The application-specific integrated circuit (ASIC) is at the other end of the spectrum. It is a processor designed specifically for its intended application [9], containing only the blocks that are necessary for optimal operation, including CPU, GPU, memory, etc. The development of such processors is expensive, time consuming and resource intensive, but they offer extremely high performance with low power consumption. The Tensor Processing Unit (TPU), for example, is an accelerator developed by Google specifically for machine learning in neural networks [154].

ARM (advanced RISC machine) is a family of reduced instruction set computing (RISC) architectures for computer processors. This type of processor is characterized by its energy efficiency, among other things, and is found especially in mobile devices such as cell phones or tablet PCs. Special ASIC components can be integrated on ARM processors, e.g. components that perform crypto-mining or vector units to perform operations on vectors. Therefore, they can also be found in supercomputers. Special ARM CPUs based on the SIMD principle (see chapter 2.1) are the basis of the number one of the TOP500.

More and more often, these types of processors are merging, so that FPGA elements also exist on GPUs, for instance. One example is the Tesla T4 GPU from NVIDIA, which is specifically designed for AI applications and contains embedded FPGA elements for AI inference applications [45].

In this thesis we limit ourselves to the consideration of CPUs and GPUs. As already described in section 1.1, these are often present in supercomputers and have a future-oriented concept.

1.3 Requirements for numerical software

The requirements of numerical software vary depending on the application. Usually a certain accuracy is given which must be achieved, or the simulation must be calculated in a given time. Also the target platform on which the software is supposed to run or, e.g., the smallest possible memory requirement can be a prerequisite. However, efficient software should, in addition to meeting the requirements, make the best possible use of the hardware so that the runtime is as short as possible.

We distinguish between two main requirements for efficient software. Firstly, the target platform on which the software is to run and, secondly, the requirements for the accuracy of the simulation.

Target platform To write efficient software it is necessary to know on which architecture the software should run and how this architecture is designed internally. For example, software for single core processors almost always needs to use use different numerical algorithms than those for multicore processors. The memory intensity of the program and the memory size of the platform should also be discussed. It can be useful to store results temporarily instead of recalculating them at a later time. This saves calculation operations, but requires additional memory. Since memory is often the one limiting resource, this is often not possible but the other way round is more useful. Likewise, software for multi-core processors quickly generates overhead that is not necessary for single core processors. Furthermore, parallel programming requires special programming interfaces, which also differ depending on the target platform. To write programs for multi-core processors, OpenMP [127] or OpenACC [126] can be used. Both parallelize programs at the level of loops that are executed in different threads. This approach requires that all participating process threads can access to shared memory (so-called uniform memory access (UMA) and non-uniform memory access (NUMA) systems), as it is the case with multicore processors. For systems with distributed memory, other approaches such as the Message Passing Interface (MPI) [108] must be used. Systems with distributed memory are, e.g., supercomputers or clusters, where several computing nodes (comparable to separate computers) are connected to each other via network connections (e.g., Infiniband). Outsourcing calculations to a GPU can also be done with current versions of OpenMP [127] and OpenACC[126]. However, to specify an implementation more closely to the architecture of the GPU, programming interfaces such as OpenCL [81] or CUDA [120] can be used.

An alternative to MPI are task-based runtime systems. Here different calculations have to be expressed as tasks and the dependencies of the different tasks have to be described by a graph. This graph is then used to determine which tasks can be executed in parallel [48]. For instance, it can be used for the parallel solving of sparse linear system of equations [91].

Another alternative for parallel systems is the programming model partitioned global address space (PGAS). Here each processor is assigned one address space of a global memory address space as local memory. Nevertheless each processor can access every memory range, where the local memory can be accessed way faster than the memory of other processors [8]. Existing implementations almost exclusively map the PGAS model to MPI-3 features.

Objectives of parallel numerics The goal of parallel programming is that the implementation scales strongly and weakly. *Strong scaling* means that the number of processors increases, whereas the problem size remains the same. If the number of processors is doubled, the runtime of the program should be halved in the best case, which is called ideal scaling.
Weak scaling means that the problem size is scaled by the same factor as the number of processors. If the number of processors is doubled and the problem size also increased by a factor of two, the runtime should ideally remain identical. Besides these two scalabilities of the implementation there is also the *numerical scalability*. It measures the scalability if the problem size is changed but the number of computing resources remains the same. Ideal numerical scalability is given, if the problem size is doubled and the runtime increases only by a factor of two, while the number of computing resources remains constant. It measures the impact when the problem size is changed independently of the number of resources. However, this is not the case with many algorithms. If Gaussian elimination is used to solve a sparse linear system of equations, it is far from ideal numerical scalability, because it requires $\mathcal{O}(n^3)$ operations (where n is the number of unknowns). If instead an iterative method, like the Gauss-Seidel method, can be used, it is possible to improve the numerical scalability significantly. The Gauss-Seidel method requires only $\mathcal{O}(n)$ operations per iteration (in case of a sparse linear system of equations), so that the cost grows linearly (assuming that the number of iterations is independent of the problem size).

But this assumption is usually not fulfilled. We still consider the solution of a sparse linear system of equations. Most Krylov subspace methods behave like $\mathcal{O}(n)$ operations per iteration. However, the number of operations depends on the problem size and behaves like $\mathcal{O}(n)$. Thus, there is a quadratic growth of the effort and therefore the numerical scalability is not optimal. An alternative would be a multigrid method, where the number of iterations is independent of the problem size [136]. Here, the effort scales ideally with $\mathcal{O}(n)$.

The numerical scalability is indirectly hidden in the strong scalability as well, because the problem size per computational resource changes, if the number of computational resources changes, while the problem size remains unchanged.

A further goal is to achieve the peak performance of the hardware. Peak performance describes the theoretical maximum performance to be achieved and is measured in FLOP/s (floating point operations per second) for the compute power, in GB/s for communication and in nano- and microseconds for communication and memory latency. However, in practice it is difficult to reach the theoretical maximum as described in section 1.3.

In this thesis we focus on the development of efficient software for small clusters up to supercomputers. For this we use OpenMP, MPI and CUDA. We will also analyze our software for weak and/or strong scalability.

Simulation accuracy Numerical software simulates physical phenomena or other processes that take place in the real world and are therefore able to serve as predictions of real processes. Their results must therefore show the same behavior as if the process itself had happened in the real world. In other words the numerical solution should have an acceptable difference to the exact solution or to experimental evidence. On the other hand, the problem must be solved in finite time. Basically a program needs more runtime the higher the accuracy of the simulation is, if the numerical methods remain the same. In the scope of this thesis, the error of a numerical simulation (simulation error) consists of the model error and the numerical error [72].
The model error comprises the following errors:

- Idealization/model error: Each process to be simulated needs to be transferred into a mathematical model first. In this mathematical model description, simplifications are usually made, so that e.g. negligible phenomena are not simulated, or non-linear processes are linearized. The differences to the real model resulting from these simplifications are called idealization or model errors.

- Data/experimental error: Each model requires input parameters, such as material parameters or an input signal. These input parameters are determined for example by experimental tests or by empirical investigations. However, since these are also subject to errors, no exact input parameters can be obtained. Likewise the input parameters are only available in a certain accuracy, because they have to be converted into finite arithmetic for the computer.

The numerical error is influenced by the following aspects:

- Truncation/discretization error: The mathematical models are solved with numerical methods. For this purpose infinite processes are usually replaced by finite ones. For example, the determination of a derivative is replaced by difference quotients. The deviation of these finite processes to the exact ones is called truncation or discretization error.

- Termination error: Infinite algorithms must be terminated after many iterations. While $\sin(x)$ can be represented exactly as an infinite sum $\sin(x) = \sum_0^\infty \frac{x^{2n+1}}{(2n+1)!}$, but this sum must be terminated after a certain number of operations. The difference between the value after this iteration and the exact solution is called termination error.

- Rounding error: On the computer system, all calculations must be carried out on a finite-precision floating point arithmetic. Therefore, arithmetic operations may not give the exact result. The difference is known as the rounding error.

In this thesis the numerical errors only play an indirect role. However, in chapter 3, the numerical error is reduced by choosing the RK4IP method in comparison to the Split-Step Fourier method, or rather the reduction of the error with the corresponding spatial discretization [33]. Furthermore, in chapter 5 the idealization error is minimized by taking further physical properties into account, and in section 5.3.3 by introducing a penalty term.

Implementation/method selection There are several methods to reduce the runtime. A simple possibility is to simplify the model description, e.g., by neglecting physical phenomena that do not have a significant effect on the scientific outcome. However, this leads directly to an increase of the idealization error. The discretization can also be coarsened, e.g., by reducing the number of degrees of freedom or by using numerical methods of low order. As a consequence, this leads directly to an increase of the discretization error. However, these two possibilities only reduce the number of operations the method needs. In this work we are interested in increasing the FLOP/s by using parallelism.

The target should always be to reduce the runtime as far as possible without increasing the numerical error. Nevertheless, numerical methods should be used and developed that require as few operations as possible and scale well. The most important point here is the scalability. For example, to solve a linear system of equations $Ax = b$, the classical Gaussian elimination, which requires $\mathcal{O}(n^3)$ operations, should not be used. In many cases it is possible to use iterative solvers, which scale much better and need only $\mathcal{O}(n)$ operations per iteration. For a solver whose number of iterations is independent of the problem size, this results in $\mathcal{O}(n)$ operations. In general, algorithms should be used which have a good numerical scalability as described above. If the hardware is used optimally, a further increase in terms of FLOP/s can only be achieved by using more (or faster) hardware. Therefore, the corresponding numerical methods must be parallelizable, which is not necessarily trivial. The following simple example is not trivial and can be easily parallelized.
Example: The goal is to compute $a = n_1 + n_2 + \ldots + n_N$, the sum of N data items. Therefore, we need $N - 1$ operations and a serial CPU implementation needs $\mathcal{O}(N)$ clock cycles for a linear straight forward approach. This problem is not trivial to parallelize for different reasons. The operations depend on each other since the second addition needs the output of the first one and so on. Furthermore, all operations need to write to the same address a, that means that we have a race condition. One way to parallelize this sum is the following: Instead of adding one date after the other, different pairs are always added in parallel. To illustrate this, let us look at the following example $a = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$ (see figure 1.2). Since '+' is both commutative and

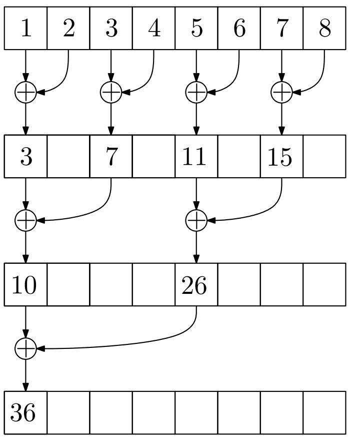


Figure 1.2: Parallelization idea of the reduction sum.

associative (at this point we neglect the floating point arithmetic), we can rewrite the sum to $a = ((1+2)+(3+4))+((5+6)+(7+8))$. Now, we can do $(1+2)$, $(3+4)$, $(5+6)$ and $(7+8)$ parallel and we save the solution in auxiliary variables. Then, we can do the next adds parallel. After that, we only need one last addition and have the solution. With this parallel approach we need 3 instead of 7 clock cycles, or $\mathcal{O}(\log_2(N))$ instead of $\mathcal{O}(N)$ in general, using $N/2$ parallel compute units.

Another way to reduce the runtime without changing the modeling and the numerical method is to adapt the implementation to the hardware used, which is the core of this thesis.

As another example, we consider a matrix-matrix multiplication, which is much faster on a GPU by splitting it to many small matrix multiplications. In this case we can use the fast access to the on-chip so called shared memory and additionally the reuse of data leads to an acceleration.

Example: We discuss the matrix multiplication of two matrices $A, B \in \mathbb{R}^{N\times N}$ and the solution matrix $C \in \mathbb{R}^{N\times N}$. For a simpler illustration we consider here an example of the size $N = 4$ with

$$A \cdot B = \begin{pmatrix} 5 & 2 & 6 & 1 \\ 0 & 6 & 2 & 0 \\ 3 & 8 & 1 & 4 \\ 1 & 8 & 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 7 & 5 & 8 & 0 \\ 1 & 8 & 2 & 6 \\ 9 & 4 & 3 & 8 \\ 5 & 3 & 7 & 9 \end{pmatrix} = \begin{pmatrix} 96 & 68 & 69 & 69 \\ 24 & 56 & 18 & 52 \\ 58 & 95 & 71 & 92 \\ 90 & 107 & 81 & 142 \end{pmatrix} = C$$

To compute one entry for the full matrix, we compute 'row times column' and it costs $N-1$ additions and N multiplications.

$$c_{11} = \begin{pmatrix} 5 & 2 & 6 & 1 \end{pmatrix} \cdot \begin{pmatrix} 7 \\ 1 \\ 9 \\ 5 \end{pmatrix} = 5\cdot 7 + 2\cdot 1 + 6\cdot 9 + 1\cdot 5 = 35 + 2 + 54 + 5 = 96$$

This is easy to parallelize, because every entry of the solution matrix C can be computed in parallel. However, we cannot influence which matrix entries will be read at the same time to reduce reading times. One way to overcome this issue is to split the matrices into submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 5 & 2 \\ 0 & 6 \end{pmatrix} & \begin{pmatrix} 6 & 1 \\ 2 & 0 \end{pmatrix} \\ \begin{pmatrix} 3 & 8 \\ 1 & 8 \end{pmatrix} & \begin{pmatrix} 1 & 4 \\ 5 & 6 \end{pmatrix} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 7 & 5 \\ 1 & 8 \end{pmatrix} & \begin{pmatrix} 8 & 0 \\ 2 & 6 \end{pmatrix} \\ \begin{pmatrix} 9 & 4 \\ 5 & 3 \end{pmatrix} & \begin{pmatrix} 3 & 8 \\ 7 & 9 \end{pmatrix} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Computing the upper left matrix C_{11} requires the matrices A_{11}, A_{12}, B_{11} and B_{21} and can be written as

$$\begin{aligned} C_{11} = A_{11}B_{11} + A_{12}B_{21} &= \begin{pmatrix} 5 & 2 \\ 0 & 6 \end{pmatrix} \begin{pmatrix} 7 & 5 \\ 1 & 8 \end{pmatrix} + \begin{pmatrix} 6 & 1 \\ 2 & 0 \end{pmatrix} \begin{pmatrix} 9 & 4 \\ 5 & 3 \end{pmatrix} \\ &= \begin{pmatrix} 5\cdot 7 + 2\cdot 1 & 5\cdot 5 + 2\cdot 8 \\ 0\cdot 7 + 6\cdot 1 & 0\cdot 5 + 6\cdot 8 \end{pmatrix} + \begin{pmatrix} 6\cdot 9 + 1\cdot 5 & 6\cdot 4 + 1\cdot 3 \\ 2\cdot 9 + 0\cdot 5 & 2\cdot 4 + 0\cdot 3 \end{pmatrix} \\ &= \begin{pmatrix} 35+2 & 25+16 \\ 0+6 & 48+0 \end{pmatrix} + \begin{pmatrix} 54+5 & 24+3 \\ 18+0 & 8+0 \end{pmatrix} = \begin{pmatrix} 37 & 41 \\ 6 & 48 \end{pmatrix} + \begin{pmatrix} 59 & 27 \\ 18 & 8 \end{pmatrix} = \begin{pmatrix} 96 & 68 \\ 24 & 56 \end{pmatrix} \end{aligned}$$

If we divide the matrices into submatrices $A_{ij}, B_{ij}, C_{ij} \in \mathbb{R}^{N/M \times N/M}$, it costs $M(N/M) + M = N + M$ additions and $M(N/M) = N$ multiplications. So, this approach requires M additions more than the standard 'row times column' approach. In practice, the size of submatrices N/M is chosen instead of the number of submatrices in one direction M. A typical choice on GPUs for N/M is 32. For a matrix with size $N = 2^{14}$ this would result in 512×512 submatrices of size 32×32. This would mean, that 512 additional additions per entry have to be done, and thus a total of $2^{28} \cdot 512$ additional additions have to be done. Nevertheless, this approach can ensure that all entries of C_{11} will compute in parallel , so that the data of A_{11}, A_{12}, B_{11} and B_{21} can be reused. In practice, all calculations are first completed with A_{11} and B_{11} and then all calculations with A_{12} and B_{21}, to achieve the best data reuse, leading to a much faster approach than the standard one, especially on GPUs.

Efficiency restrictions It is not always possible to reach the the peak performance of the hardware. Peak performance describes the theoretical maximum performance to be achieved and is measured in FLOP/s. It is desirable to come close to this performance, because otherwise there are still unused resources available that could be exploited.

We consider the supercomputer Piz Daint Cray XC50, which has a peak performance of 27.1 PFLOP/s and consists of 5704 computing nodes, each containing an Intel Xeon E5-2690v3 and an NVIDIA Tesla P100 [153]. The peak performance results from the sum of the peak performance of all P100 GPUs. This means that a software must be executable on the GPU and able to use the parallelism of the GPUs as best as possible to achieve this peak performance. If we calculate the peak performance of the CPUs in this system, we get a peak performance of 2.8 PFLOP/s that is almost 10 times lower [49].

However, there are also restrictions for software that was written exactly for the target architecture. High performance applications depend on high utilization of bandwidth and computing resources. They are usually limited by either memory or computing speed [77]. Memory-bound applications reach the limits of system bandwidth, while compute-bound applications exhaust the computing capacity of the processor. A common approach in computer architecture and processor design to alleviate the memory-bound problem is to employ even larger hierarchies of fast, on-chip cache memories. Hierarchical caches are standard components of modern processors that are used to increase memory bandwidth and reduce average latency, especially when successive memory accesses are spatially local. This problem also exists when using GPUs if data needs to be copied to and from the GPU.

1.4 Software packages used in this thesis

This work is about method development for challenging PDE problems driven by three different applications. For each of these use cases we use and extend different software to solve these problems. For the simulation of nonlinear signal propagation in multimode fibers we use a custom implementation, which was co-developed by Marius Brehler at the Institute for High Frequency Technology at TU Dortmund University. This software is written in C++ and is parallelized with the help of OpenMP, CUDA and MPI. The CUDA and MPI improvements presented in chapter 3 are also included in the software.

For the particle-based simulation of flow in porous media presented in chapter 4 we use our in-house software called `hoosph`. It is based on the software package hoomd-blue [13] and was extended with functions to use the SPH method [140] by the group of Prof. Steeb at the Institute of Mechanics at the University of Stuttgart.

A customized implementation was also used for the seismic waveform modeling and inversion simulations described in chapter 5. This software is called SOUNDVIEW and is mainly maintained by Vadim Monteiller at the 'Laboratoire de Mécanique et d'Accoustique' in Marseille. This is a finite difference solver implemented in C++ and uses MPI and CUDA for parallelization.

1.5 Thesis contribution

The first, more general, main contribution of this work is the development of efficient PDE simulation software for smaller clusters and large supercomputers. For this purpose, we derive exemplary improvement techniques on the basis of three different applications, which can be transferred and adapted to other similarly 'unstructured' PDE problems. The second, more specific, main contribution of this work is the extension from elastic to viscoelastic materials for the inversion of seismic wave propagation, as well as an improved solver method which yields better results, especially for disturbed data. In the following, we list the different contributions in more detail and refer to the respective chapters in which they are described:

- In chapters 3 and 4 implementation techniques are presented which lead to a large performance benefit. These techniques are discussed in the sections 3.4.3 and 4.4.3. These implementation techniques can also be applied or adapted to other PDE problems. In particular, many of the more general techniques have been applied in chapter 5.

- In chapters 4 and 5 modified/improved algorithms, data structures and numerical schemes are presented. In section 4.4.3 a modified neighbor search algorithm is presented and thereby in particular an SPH implementation based on a pairs neighbor list instead of a Verlet list. This leads to a significant performance gain on modern GPU hardware. In section 5.3.3 a (constant) Q factor approximation, which is based on an idea of Fichtner and van Driel [54], is modified so that a higher accuracy is achieved. Furthermore, in section 5.6.4 a modified inversion method is presented, which can provide a lower model misfit especially for problems with disturbed data.

- In chapter 5, the inversion of seismic wave propagation is extended for viscoelastic materials. The influence of viscosity can have a large impact on the inversion [52, 90], so the accuracy of the inversion increases when viscosity is included. The approach presented directly inverts the Q factors and thus can be applied to non-constant Q factors as well.

- In the pursuit of this thesis, the implementations have been incorporated into various software packages, and are partly available open-source or are planned to be made accessible. Parts of the GPU and MPI implementations of the software used in chapter 3 were applied in the context of this thesis. Within chapter 4 the GPU implementations for the SPH module for HOOMD-blue were done, as well as contributions to the MPI and parallel I/O implementation. In addition, a further neighborhood search algorithm and a pairs list were implemented. The GPU implementation, as well as the implementation of all functions for the viscoelastic wave modeling and inversion, have been done in the process of chapter 5 and added to the software SOUNDVIEW.

1.6 Thesis outline

This thesis is written in such a way that all chapters are self-contained, i.e., each chapter provides its own motivation and summary. Nevertheless, the chapters all build upon each other, so that early chapters offer insights into later ones, and later chapters refer to aspects already described in earlier chapters. In the following we give an overview of the individual chapters and refer to the respective introductory sections for a more detailed structure.

In chapter 2 we give an insight into GPU programming, starting with the basic structure of GPUs and the difference to CPUs. Then, in section 2.2 we present the challenges that an efficient GPU implementation must overcome. We also show simple examples of how an implementation can be improved and what needs to be taken into account to achieve efficient programming. The chapter is completed with section 2.3, which gives an insight into multi-GPU programming and shows which possibilities exist to implement it.

The challenges of efficient GPU implementations are then examined in more detail in chapters 3 and 4 using two application examples. General and specific techniques for these applications are shown in detail. In chapter 3 the simulation of nonlinear signal propagation in multimode fibers is considered. First, section 3.1 motivates the topic and describes why an efficient realization on multi-GPU systems is desirable. Then, in section 3.2 the modeling method of choice is presented, followed by the numerical approximation in section 3.3 in which the used algorithm is also presented. In section 3.4 we discuss the different implementations on the CPU and the GPU, explain the challenges and present an efficient GPU implementation. Afterwards, in section 3.4.4, we discuss different approaches to multi-GPU implementation, as well as present the implementation techniques. Next, section 3.5 compares the different approaches. The chapter is concluded in section 3.6 with a summary of the improvements our changes have achieved and the GPU challenges we have addressed.

The second application is the particle-based simulation of flow in porous media which is the subject of chapter 4. Section 4.1 motivates why an efficient realization is necessary and in which scenarios such a simulation is used. Next, we give the mathematical model in section 4.2, followed by the numerical approximation using the SPH method in section 4.3. Here we go into detail about the discretization and end with a feasible algorithm. In the next section 4.4, we describe the challenges of the GPU implementation, as well as the implementation in HOOMD-blue, and present three different possible improvements that can be applied incrementally. These improvements as well as the scalability of our implementation are validated in section 4.5. We conclude the chapter in section 4.6 with a summary of what improvements our changes have achieved and the GPU challenges we have addressed.

In chapter 5 we consider the modeling of seismic wave propagation as well as the inversion. Due to the improvement possibilities presented in the previous chapters, we can now increase the accuracy of the numerical modeling in this chapter, i.e. we include more physical parameters, and thus make the problem more complex. However, such complex problems can usually only be solved if the implementation is efficient, otherwise the problem cannot be solved in a reasonable amount of time. Section 5.1 highlights why an efficient realization is necessary and in which scenarios such a simulation is used. Then in section 5.2 we introduce the mathematical model, where we derive the viscoelastic wave equation using a rheological model. The resulting forward model is presented in section 5.3, but introduces some drawbacks that we eliminate in the following, based on an idea of Fichtner and van Driel [54]. Furthermore, in this modification we add a kind of penalty term to further improve the method. In section 5.4 we then present the inversion, using the adjoint state method for computing the derivatives, which is also presented in this chapter. Afterwards, section 5.5 presents an algorithmic implementation, and finally a possible FWI algorithm is presented. The improvements produced by the changes shown are investigated in section 5.6. First, a simple example is used to investigate whether a viscoelastic inversion is necessary and whether it yields a significant difference compared to a pure elastic inversion. Then, different regularization methods are compared, both for noisy and unnoisy data. Since even the best regularization method yields much worse results with respect to the model misfit than for unperturbed data, we present another idea based on the best regularization method that yields better model misfits. We then apply this idea to the unperturbed data as well, resulting in faster convergence. Next, we investigate the impact of the multiscale approach on viscoelastic modeling. These findings are then applied to the Marmousi example, a common example in geophysics that resembles a real structure. Finally, we consider the scalability of the implementation. The chapter is concluded in section 5.7 with a summary of which improvements our changes have achieved.

In the last chapter, 6, we summarize all findings of this work and give an outlook on further open questions.

2

GPU Computing

This chapter describes the basic structure of a GPU and compares it with that of a CPU. Afterwards the challenges for implementing software for the GPU are described. The terminology of the CUDA programming language is introduced and used. It also describes techniques for improving a GPU implementation and the existing memory hierarchy. Basic knowledge is assumed or only covered very briefly in this chapter. This chapter is based on a tutorial that the author has co-presented in 2017 [67] and was updated and extended for this work. For a more detailed study of programming in CUDA, see [67, 165].

2.1 Specifics of a graphics processing unit (GPU)

Initially, GPUs were developed to create images for computer graphics and video game consoles. This means, a GPU is designed for fast simultaneous rendering of high-resolution images and video. Rendering these images involves many operations (for each pixel), but always the same operations with different data are performed for each pixels. It therefore has many parallel processing units that can perform these calculations simultaneously.

The difference to a CPU is simply stated, that everything that makes a single instruction fast is eliminated or greatly reduced, which concerns especially the caches and hard-wired control logic. In contrast, the number of ALUs is increased (see figure 2.1). Due to the simple control logic all ALUs can only perform the same calculations. This is known as single instruction multiple data (SIMD).

In order to further increase the degree of parallelism, this simplified compute unit is cloned several times. This simplified comparison between CPU and GPU is shown in figure 2.1. A CPU can thus process a sequential instruction much faster. The clock frequency of the Intel Xeon Platinum processor 8256 with 3.8 GHz is more than twice as fast as the clock rate of a NVIDIA V100, one of the latest professional GPUs, with a clock frequency of 1.53 GHz. However, the GPU's fine granularity allows many more operations to be performed simultaneously.

On the contrary, a multi-core CPU can run different programs on all of its cores (known as multiple instruction, multiple data), whereas on a GPU it is always necessary to use SIMD to take advantage of the fine granularity and to obtain acceleration. However, not all GPU cores have to process the same instruction, only a so-called warp. This is explained in section 2.2.

A CPU can never be completely replaced by a graphics processor: A GPU complements the CPU architecture by allowing repetitive calculations within an application to run in parallel while the main program continues to run on the CPU. The CPU can be considered as the task manager of the entire system, coordinating a wide range of general purpose computing tasks, while the GPU performs a narrower range of more specialized (usually mathematical) tasks. By harnessing the power of parallelism, a GPU can do more work in the same amount of time compared to a CPU, what we call high throughput. CPUs have large and wide instruction sets that manage every input and output of a computer, which a GPU cannot do.

Since GPUs can perform parallel operations on multiple data sets, they are often used for non-graphical tasks such as machine learning and scientific computing. With thousands of processor cores running simultaneously, GPUs enable massive parallelism, with each core focused on performing efficient computations.

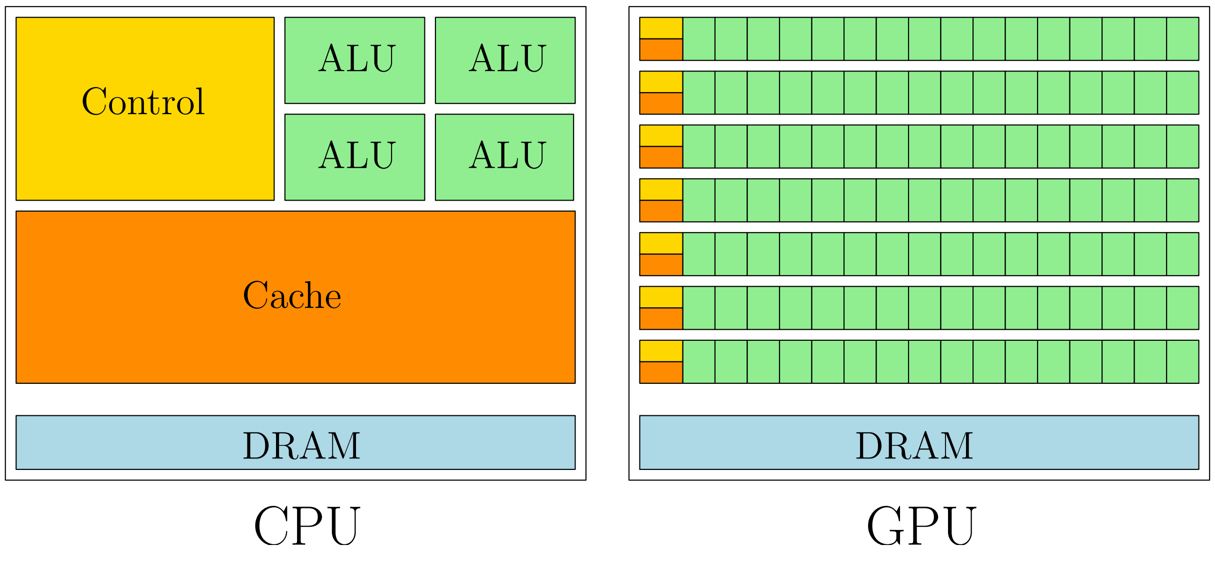


Figure 2.1: Schematic difference between CPU and GPU (inspired by NVIDIA CUDA Programming Guide [120]).

2.2 Challenges of GPU computing

Calculations on the GPU must be done according to the SIMD principle, i.e., the same operations must be performed with different data. These operations can then be performed in parallel. A simple example is the addition of two vectors $v = (v_1, \ldots, v_N)$ and $w = (w_1, \ldots, w_N)$ of the dimension $N \in \mathbb{N}$. The vector addition $v+w = (v_1+w_1, \ldots, v_N+w_N)$ consists of N additions (single instruction) that operate on different data (multiple data).

Basically, it is necessary that the work to be offloaded to a GPU is complex and large enough. On the one hand the degree of parallelism must be high, because only in this case the GPU has a throughput advantage over the CPU. Furthermore, enough operations must be done with this data. Obviously, offloading calculations to the GPU is only worthwhile if copying the data to and from the GPU and doing the calculations takes less time than doing the calculations on the CPU. If we assume a bandwidth of 16 GB/s for the transfer and neglect latencies, $2 \cdot 10^9$ double precision (64 Bit) floating point values per second (2 Giga float/s) can be copied from the CPU to the GPU and simultaneously in the other direction. This means that we need two times 0.5 seconds to copy 10^9 double precision values from the CPU to the GPU and back to the CPU. A V100 has a peak performance of 7 TFLOP/s for double precision. Therefore at least 7000 operations are necessary with each transferred date, so that the calculation on the GPU takes longer than copying back and forth.

It is important to note that if possible the copying of data should always be asynchronous, meaning that the copying of data can be done in parallel to do computations with different data. At the top of figure 2.2 a typical offloading to the GPU is depicted without asynchronous copying. First, all data needed for the calculation on the GPU must be copied to it. Then the calculations can begin. The data can not be copied back until the calculations are completed. At the bottom of figure 2.2 the advantage of asynchronous copying is shown. All the data are divided into smaller data packets on which independent calculations can pass. Now these individual packages are copied asynchronously. As soon as the first package is completely copied, the calculations for these data can start already. The copying of the data of the next packages now takes place in parallel to the calculations of previous packages. The same can be applied for copying back.

Obviously, the overall goal is to hide as many data transfers as possible, i.e., to let them take place in parallel to calculations.

To understand further challenges in GPU programming it is necessary to explain the concept of threads, blocks and grids.

2.2.1 CUDA concepts

In order to explain GPU implementations more easily we want to define some basic concepts. A *thread* is a simplified view of how a compute unit in modern processors executes a sequential program. It consists of the program's code, the specific point in the

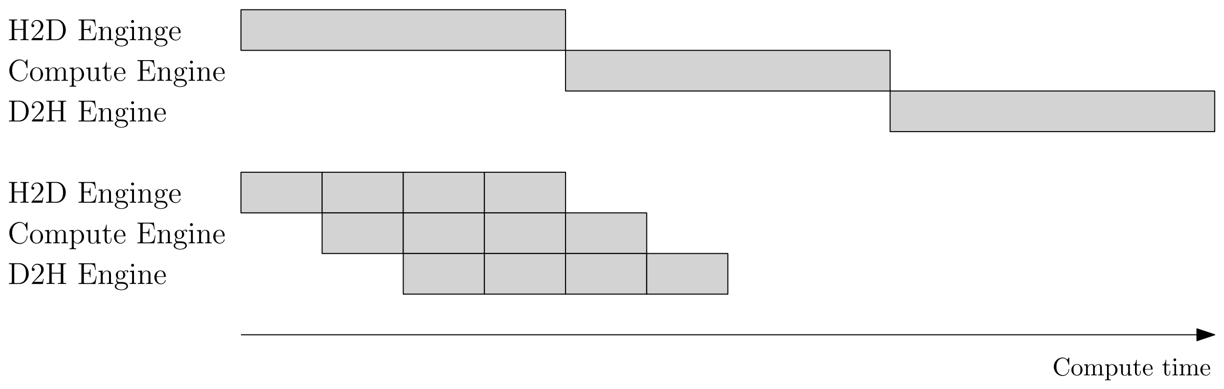


Figure 2.2: Asynchronous copying. Overlapping of data copying and computation.

code that is currently executing, and the values of its variables and data structures. The execution of a thread is sequential for a user. We consider again the addition of the two vectors $v = (v_1, \ldots, v_N)$ and $w = (w_1, \ldots, w_N)$ of the dimension $N \in \mathbb{N}$. In this case we have N threads, where thread i would perform the addition $v_i + w_i$. Due to the SIMD approach on current NVIDIA GPUs, the threads are executed in groups of 32 threads, which is called *warp* (on current AMD GPUs the threads are executed in groups of 64 threads, which is called wavefront). Therefore, threads within a warp must follow the same execution trajectory. All threads must execute the same instruction at the same time. This can lead to warp divergences as described in the next section.

Several threads are grouped together to form a *block* and can be considered as a virtual core. A block gets scheduled to a core and stays until completion.

All these blocks are further combined into a *grid* to execute a kernel. Whereas a *kernel* is nothing else than a scalar code executed by many threads in parallel.

2.2.2 Warp divergence

The most common code construct that can cause warp divergence is branching for conditionals in an if-else statement. If some threads in a single warp evaluate to ‘true’ and others to ‘false’, then the ‘true’ and ‘false’ threads will branch to different instructions (see figure 2.3). Some threads want proceed to the ‘if‘ instruction, others to the ‘else’. In this case, a warp must go through both cases. All threads do the same operations, both those in the ‘true branch’ and those in ‘false’ branch. Though the threads which have evaluated the ‘if’ condition positively reject the result of the ‘false’ branch and vice versa.

In order to use the full power of a GPU and get as close as possible to its peak performance, it should always be tried to generate as little warp divergence as possible. Warp divergences that contain only a few instructions in the respective branches are less critical than those that contain many instructions, because the more instructions a warp divergence contains the longer other threads are idle.

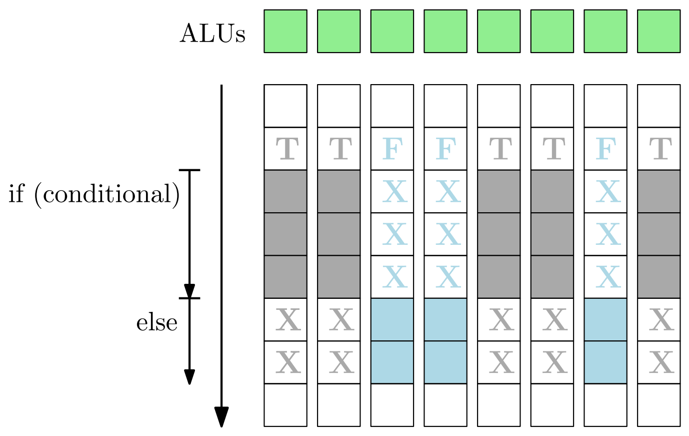


Figure 2.3: Simplified warp divergence for only eight threads.

2.2.3 Race conditions

Another challenge is when different threads want to write to the same address, or increase the value on the same address. Fortunately, race conditions are easy to avoid in CUDA. An atomic operation is capable of reading, modifying, and writing a value back to memory without the interference of any other threads, which guarantees that a race condition will not occur. Atomic operations in CUDA generally work for both shared memory and global memory. Atomic operations in shared memory are generally used to prevent race conditions between different threads within the same thread block. In global memory atomic operations are used to prevent race conditions between two different threads regardless of which thread block they are in.

The worst scenario would be if all 32 threads in a warp would write to the same address. In this case all threads would try to get write access to the address at the same time, but only one thread would be able to write, and the remaining 31 threads would be idle. In the next cycle, the remaining 31 threads would try to write to the address again, and only one thread would be able to write its value to the address again. So again 31 threads would be idle. One thread because it has already finished its operation and waits until the remaining threads of the warp have finished their operations, and the remaining 30 threads because they are waiting to write to the address. So it would take 32 times longer than writing to separate, linearly allocated memory locations.

However, there are special cases that can be avoided by using Warp-Level Collectives. Supposed we want to count the number of fluid and solid particles. To do this, each thread loads a particle, checks if it is a fluid or solid particle and increments the corresponding counter by one. This would lead to a warp divergence and also to all threads in a warp writing to the same address. This can be avoided with warp-level collectives. For this purpose it is checked how many threads are active in the warp, which are exactly the threads that want to write to the same address. Then a selected thread increases the address with an atomic operation by the value of the active threads. In this way, atomic operations are needed only in one instead of many of the active threads [3].

2.2.4 Latency Hiding

Overall, to load data from the memory is much more time-consuming than an arithmetic operation. Therefore it is important to hide these latencies by operations with already loaded data, which is called latency hiding. One possibility to hide latencies that was already introduced is asynchronous copying. It hides the latency that occurs when copying data from CPU memory to GPU memory. In addition, a GPU contains memory of different speeds and certain properties. As with CPU memory, a consecutive memory access is more efficient, since several necessary data are made available by using cache lines with one loading process. These types of memories are described in the following section. The correct use and utilization of these memory types allows to hide many latencies.

Another way to hide latencies is the right choice of number of blocks and threads per block. In general, a higher number of blocks is better, because this way more blocks are available for swap as soon as latencies occur. The number of blocks is of course limited by the resources like register file, L1 cache, shared memory, etc. If one resource is exhausted, it is not possible to schedule more blocks. Therefore, a good way to get a good indication of the correct size of the blocks is the NVIDIA CUDA Occupancy Calculator [121]. For more details and strategies we refer to [67, 165].

2.2.5 Different types of memory

In addition to hiding latencies, it is also recommended to reduce them. This can be achieved by using special types of memory such as shared memory, but especially by using the respective memory types in the best possible way. To achieve this, we will introduce different types of memory in the following and explain how they should be used to achieve the lowest possible latencies.

Global memory The access to the global memory is the slowest of all memories. However, it is accessed using cache lines, so that if the global memory is accessed linearly, many cache misses can be avoided, thus reducing latency. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions.
The following example shows two different accesses to the global memory and compares them:

Example: Assume we have N particles, each with the properties density, pressure and energy. There are two different ways to store these values. Either the values are stored in an array of size $3N$, where first density, pressure and energy of particle 1, then density, pressure and energy of particle 2 and so on are stored (this is called array of structures (AoS)). Or all densities are stored first, then all pressures and finally all energies, or to store three arrays `d`, `p` and `e` with N entries each (this is called structure of arrays (SoA)).

Densities, pressures and energies would then be stored like shown in figure 2.4
Let us next look at the operation x[i]=d[i]+y[i] where x and y are two arbitrary arrays.

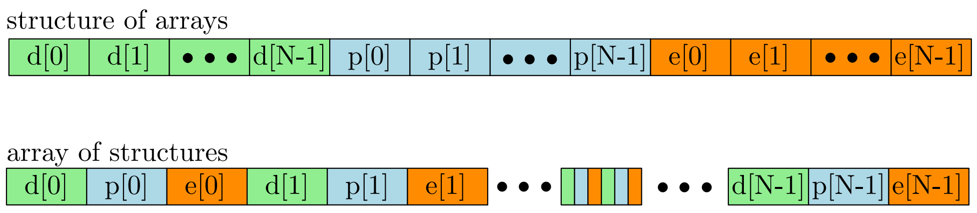

structure of arrays
d[0] d[1] • • • d[N-1] p[0] p[1] • • • p[N-1] e[0] e[1] • • • e[N-1]
array of structures
d[0] p[0] e[0] d[1] p[1] e[1] • • • • • • d[N-1] p[N-1] e[N-1]

Figure 2.4: Array of structures and structures of array data layout.

The first task is to load d[i]. In the case of AoS thread 0 reads d[0]. The cache line would automatically loads p[0] and e[0] as well, even if they are not needed. In the case of SoA, by contrast, the situation would be as follows. Thread 0 wants d[0], thread 1 wants d[1], thread 2 wants d[2] and so on. Since a cache line automatically loads these values at the same time, there are much fewer cache misses than with AoS.

Since the access to the global memory usually can not be avoided completely and the correct memory access is therefore a very important point. We consider another example in which besides the correct memory access also the reduction of warp divergence is illustrated.
Example: We consider the marix vector multiplication for sparse matrices [41]. Since many of the entries in sparse matrices are zero, there is no need to store them explicitly. There are many compressed storage formats for sparse matrices. One of them is the Compressed Storage by Rows (CSR) format, which stores non-zero entries of the sparse matrices in row order. Two arrays are used for indexing non-zero entries. The elements in the `row_ptr` array point to the first non-zero entry in each row. There are a number of rows+1 elements in this array, with the last element retained to indicate the boundary of the last row. The other array stores the column indices of the non-zero elements in row order. Figure 2.5 shows an example of CSR. A possible GPU implementation would use one thread calculating one entry of the result vector, so a thread would do the calculation 'row times column'. The pseudo code is described in algorithm 2.1.

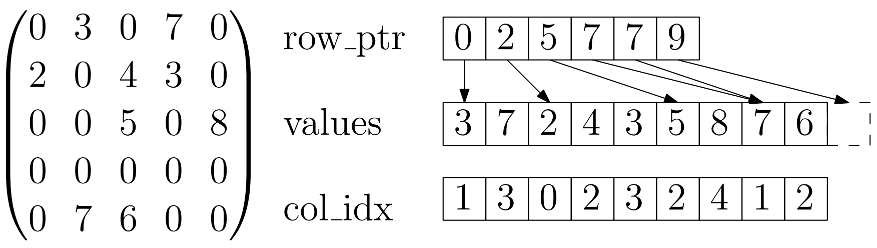

$\begin{pmatrix}0&3&0&7&0\\2&0&4&3&0\\0&0&5&0&8\\0&0&0&0&0\\0&7&6&0&0\end{pmatrix}$
row_ptr 0 2 5 7 7 9
values 3 7 2 4 3 5 8 7 6
col_idx 1 3 0 2 3 2 4 1 2

Figure 2.5: Example CSR Format.

Algorithm 2.1 (Matrix-vector multiplication (CSR-Format)).

```
1: **Input:** x, row_ptr, col_idx, values
2: **Output:** y
3: row = unique_thread_id
4: row_start = row_ptr[row]
5: row_end = row_ptr[row+1];
6: dot = 0
7: for elem=row_start, ..., row_end do
8:     dot=dot + values[elem] · x[col_idx[elem]]
9: end for
10: y[row] = dot
```

However, this approach has two disadvantages. First of all, adjacent threads execute different numbers of iterations in its for-loop. Furthermore, adjacent threads access non-adjacent memory locations.

One matrix format that solves these disadvantages is the jagged diagonal storage (JDS) format [114]. To store a matrix in JDS format, the matrix rows are rearranged in descending order according to the number of non-zeros in each row. Then, all nonzeros of the matrix are shifted to the left. Columns of this new compressed matrix are called jagged diagonals. Nonzero values of the compressed matrix are stored in an array in column order. Corresponding column indices of each nonzero value in the original matrix are written to another array. Another array is used to point the beginning indices of each jagged diagonal. In addition, the row permutation is stored in an array, with elements of the array corresponding to the rows in the compressed matrix pointing to the row number in the original matrix. It is also useful to store the number of non-zero entries for each row. Figure 2.6 shows an example of JDS and algorithm 2.2 the pseudo code for a possible GPU implementation. In the figure `perm, jd_ptr and col_idx` each represent permutation, jagged diagonal pointers and column index arrays. `nnz_row` contains the number of nonzero entries for each row.

The JDS format has advantages especially when the number of nonzero entries per row varies significantly. Sorting according to the number of nonzero entries means that the loop of adjacent threads must pass through approximately the same number of iterations. This leads to the lowest possible warp divergence. Additionally, adjacent threads access adjacent memory locations.

This difference can be seen when comparing line 7 in algorithm 2.1 with line 6 in algorithm 2.2. The loop in algorithm 2.1 runs over all non-zero entries of a row. However, since neighboring rows have a different number of non-zero entries, warp divergences always occur if not all rows processed by a warp have the same number of non-zero entries. This is different with algorithm 2.2, where rows are sorted by the number of non-zero entries, which minimizes warp divergence. The second major advantage of algorithm 2.2 becomes clear when comparing row 8 of algorithm 2.2 with row 8 of algorithm 2.1. Algorithm 2.2 provides continuous memory access for threads in a warp, whereas algorithm 2.1 does not.

$$\begin{pmatrix}0&3&0&7&0\\2&0&4&3&0\\0&0&5&0&8\\0&0&0&0&0\\0&7&6&0&0\end{pmatrix} \longrightarrow \begin{array}{ccc}2&4&3\\3&7&\\5&8&\\7&6&\end{array} \longrightarrow \begin{array}{l|c|c|c|c|c|c|c|c|c|}
\text{perm} & 1&0&2&4&3 \\
\text{nnz\_row} & 3&2&2&2&0 \\
\text{jd\_ptr} & 0&4&8&9&9 \\
\text{col\_idx} & 0&1&2&1&2&3&4&2&3 \\
\text{values} & 2&3&5&7&4&7&8&6&3
\end{array}$$

Figure 2.6: Example JSD Format.

Algorithm 2.2 (Matrix-vector multiplication (JDS-Format)).

```
1: Input: x, perm, nnz_row, jd_ptr, col_idx, values
2: Output: y
3: t_id = unique_thread_id
4: num_row_entries = nnz_row[t_id]
5: dot = 0
6: for elem=0, ..., num_row_entries do
7:     offset = jd_ptr[elem]
8:     dot=dot + values[t_id + offset] · x[col_idx[t_id + offset]]
9: end for
10: row = perm[t_id]
11: y[row] = dot
```

Constant memory The constant memory space resides physically in device memory. The constant memory space is cached and will speed up data fetch. It is only a small memory (64kB for the most GPUs) that is read only. Using constant memory instead of global memory may reduce the memory bandwidth and latency. Constant memory is also most effective when all threads access the same value at the same time.

Shared memory and bank conflicts Shared memory is on-chip, and therefore it has much higher bandwidth and much lower latency than local or global memory. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Each read or write request of the memory consisting of n addresses falling into n distinct memory banks can therefore be serviced simultaneously, resulting in a total bandwidth that is n times higher than the bandwidth of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts. Therefore, for maximum performance, it is important to understand how memory addresses are mapped to memory banks in order to plan memory requirements so that bank conflicts are minimized.

It depends on the GPU architecture how the shared memory is divided. In the following we consider GPUs with compute capability 7.x (Volta architecture). GPUs with a different compute capability may have a different partitioning, but the principle is the same.

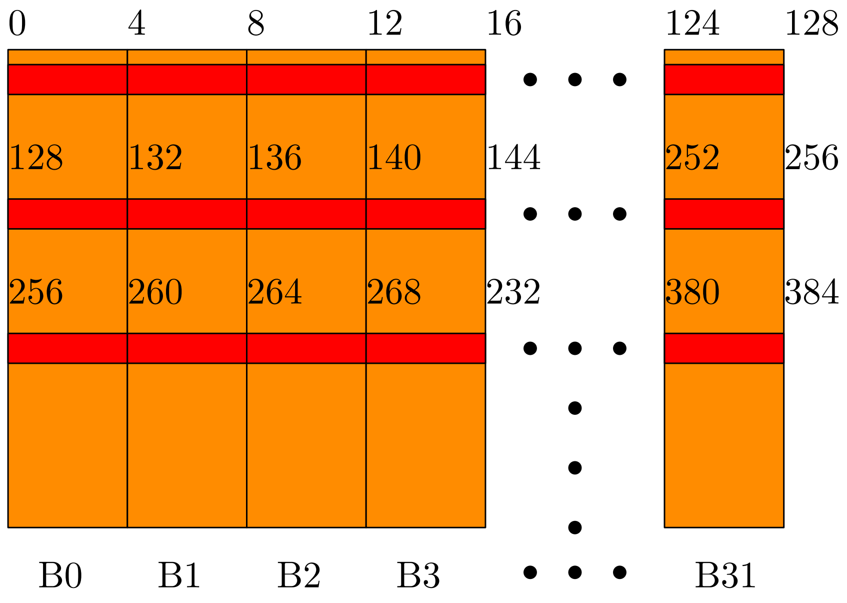


Figure 2.7: Shared memory is divided into equally-sized memory modules, called banks.

The shared memory has 32 banks that are organized such that successive 4 byte words map to successive banks (see figure 2.7). Each bank has a bandwidth of 4 bytes per clock cycle. A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 4 byte word (even though the two addresses fall in the same bank). A bank conflict occurs when at least two threads of a warp access (at least) two different 8 byte words of the same bank. Various scenarios are shown in figure 2.8. The two scenarios on the left are each conflict-free, since each thread accesses memory in different banks. The third scenario describes a two-way bank conflict, since, e.g., threads T0 and T4 access different 4 byte words in bank B0. The fourth scenario is conflict free again, because different threads only access the same 4 byte words in one bank. If a one byte word like `char` is used, threads 0, 1, 2 and 3 will store all their values in the first bank of shared memory, resulting in a bank conflict known as a 4-way bank conflict. Since 8 byte words extend over two banks and require two cycles to load anyway, they are accessed in two phases. That means that in the first phase the first 16 threads in a warp access to the shared memory, and in the second phase the other

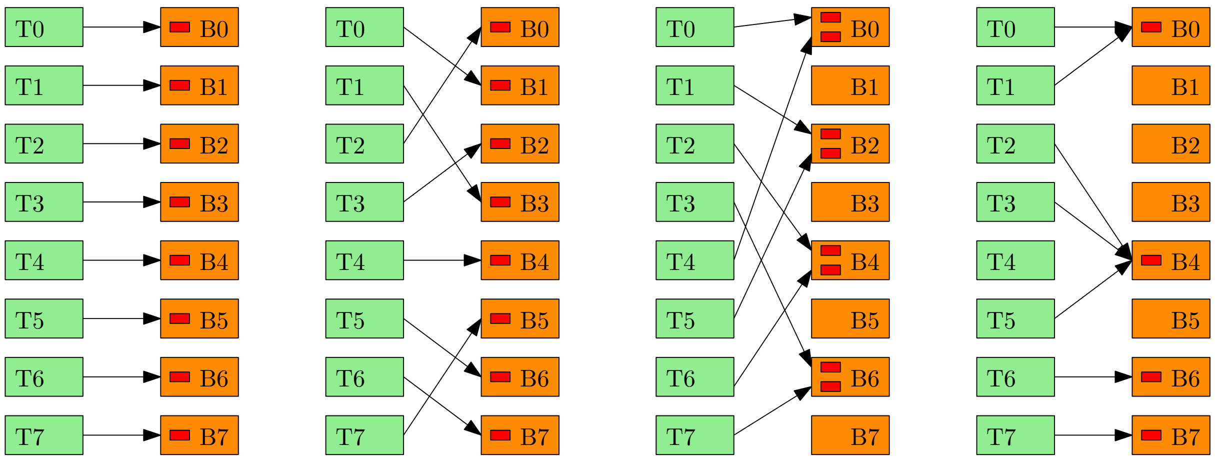


Figure 2.8: Simplified visualization of (no) bank conflicts for only 8 Threads and Banks: left and second from left: no bank conflicts, second from right: two way bank conflict, right: no bank conflicts.

16 threads. As a result, a bank conflict does not occur when a warp writes continuously to a `__shared__ double array[32]` array.

Example: To transpose a matrix the shared memory should be used. For the sake of simplicity, we use a 32 × 32 matrix. For this purpose, a `float` array of the size 32 × 32 is allocated in the shared memory, e.g. `float array[32][32]`. Thus the first column of the array falls into bank 1, the second column into bank 2 and so on. This is illustrated in figure 2.9 on the left. Since the transposition requires both row and column access, we consider both. Since all entries of a row fall into different banks, there is no conflict here. However, all entries of a column fall into exactly one bank each, so there is a 32-way conflict.

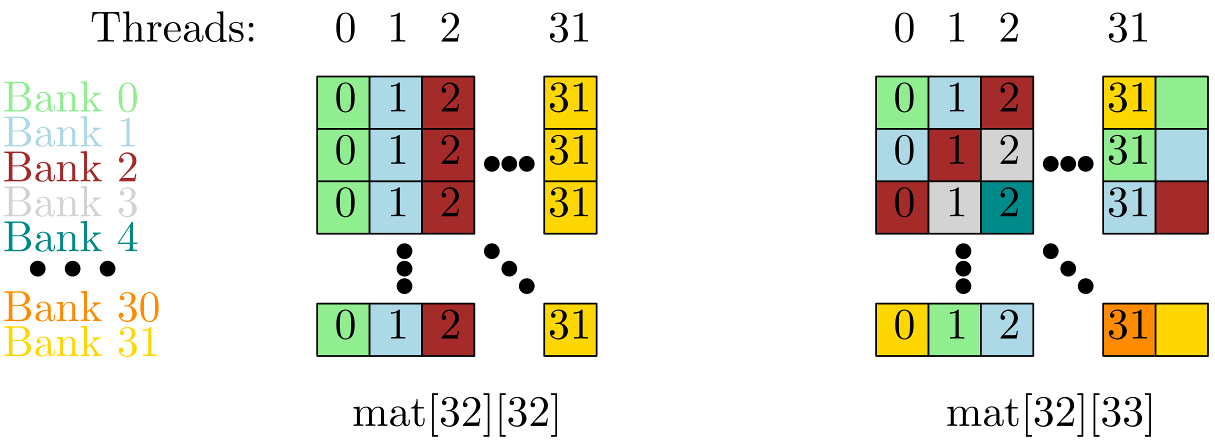


Figure 2.9: Use of shared memory for matrix transformation. left: bank conflicts for column access, right: no bank conflicts.

However, this can be easily solved by allocating a 32 × 33 array instead of a 32 × 32 array, i.e. `float array[32][33]`. The last column of the array remains unused. However, this trick causes all entries of each row, as well as all entries of each column, to fall into different banks (see figure 2.9 on the right), so no bank conflict occurs.

The respective challenges for the applications considered in the thesis can be found in section 3.4 and 4.4.2.

2.2.6 Streams

To make better use of all cores of a GPU, it is helpful to have several different applications (or kernels) running simultaneously on the GPU. This also means that different CUDA kernels can run simultaneously. This can be achieved by using streams. Kernels that are launched in different streams can be run in parallel. Suppose there are seven kernels that can actually all be run independently of each other. If no streams are used, the default stream is automatically used, and the kernels are executed one after the other (see figure 2.10 at the top). If a separate stream is used for each kernel, the kernels can be run in parallel. While this may cause the individual kernels to take longer to complete, the higher degree of parallelism is better at hiding latency, reducing the total time (see figure 2.10 at the bottom). If the default stream is used again after running kernels in different streams, this creates a barrier because the default stream is always run alone and no streams are run in parallel.

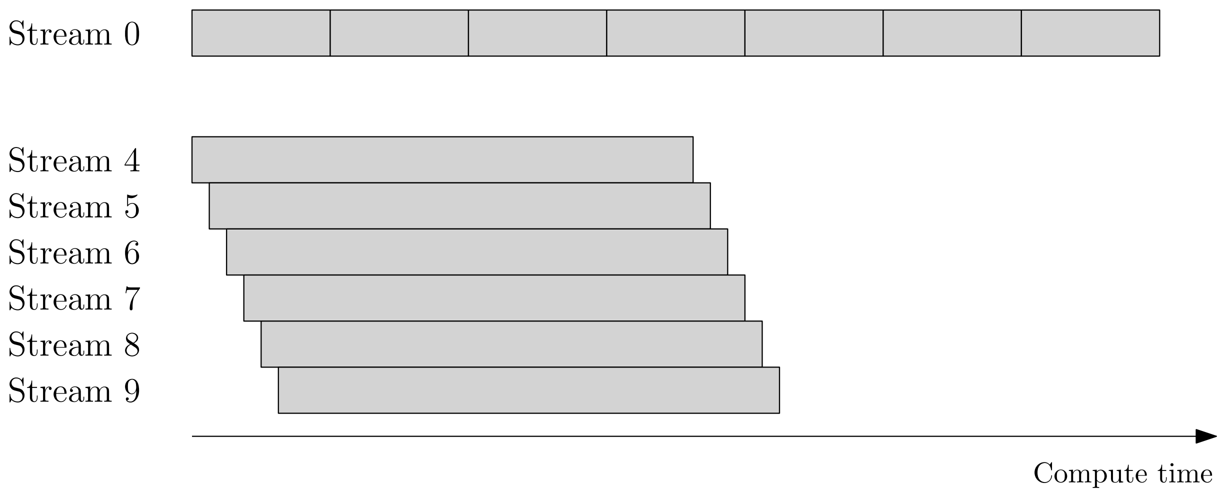


Figure 2.10: Concept of CUDA Streams. Top: use of default stream only, bottom: one stream for each work package.

2.3 Multi-GPU computing

To further increase the degree of parallelism, several GPUs can be used for a calculation. This additionally avoids the need to repeatedly copy data back and forth between CPU and GPU for applications that exceed the GPU memory. We limit ourselves to modern hardware and software. The implementation of communication can be more complicated with older versions. As with CPUs there are also two different scenarios [110]: If the GPUs are in the same nodes, the GPUs know each other and can therefore communicate directly with each other. In this case Unified Virtual Addressing is available [120], i.e., there is one address space for the entire CPU and GPU memory. Copying the data is very easy in this case, because we do not have to specify in which memory the destination or source is located. Peer-to-peer (P2P) communication is also possible in this case, i.e., data can be copied between GPUs without having to use the host memory. Bytes are transferred along the shortest PCIe path. In both cases, however, it is necessary that the GPUs are connected to the same CPU. QPI and PCIe protocols are not compatible with P2P communication. In addition to the connection via PCIe, the GPUs can also be connected via NVLINK. This technology was developed by NVIDIA to provide higher bandwidth for communication between GPUs. A single NVIDIA A100 GPU supports up to twelve third-generation NVLINK connections for a total bandwidth of 600 GB/s. This is 10 times the bandwidth of fourth-generation PCIe [120, 122]. The first version of NVLINK was introduced with the Pascal architecture and offers a total bandwidth of 160 GB/s, while the second variant of NVLINK, introduced with the Volta architecture, offers a total bandwidth of 300 GB/s [123].

If the GPUs are in different compute nodes, the communication must be done via MPI. Another possibility is the communication via NVIDIA Collective Communications Library (NCCL). More details about the communication between GPUs on different boards as well as the communication via MPI and NCCL is described in section 3.4.4.

In summary, as long as the GPUs are connected to the same CPU, or are sitting on the same board, multi-GPU programming can be implemented using CUDA without any additional tools. However, if the GPUs are located in different nodes that are connected e.g. via Infiniband, further tools such as MPI are necessary. Since this work is specialized on implementations for small clusters and large supercomputers, we restrict ourselves to implementations with MPI-like communications. However, this is not a limitation, since this implementation of course works for multiple GPUs sitting on one board.

3

Nonlinear signal propagation in multimode fibers

In this chapter the simulation of nonlinear signal propagation in multimode fibers is considered as an application. The GPU implementation is challenging because the underlying algorithm requires both row and column access to the data of a matrix. In addition, the data to be transferred between different GPUs becomes very large, so communication between GPUs is also demanding. In section 3.1 we first describe in which application areas this simulation is needed and why it makes sense to rely on GPU computing. Then, in section 3.2, we explain the physical model description. The numerical methods used are explained in section 3.3. There we first identify the challenges for an efficient implementation, followed by a description of a CPU and (multi-)GPU implementation. For communication in multi-GPU implementations different approaches are presented in section 3.4.4. Finally, the presented implementations are analyzed and compared in section 3.5. The target platforms for this application are small clusters with powerful GPUs.
The results in this chapter have been partially published in [33, 34, 35], where the author of this thesis focused on improving the (GPU) implementation.

3.1 Motivation

Today, fiber optic cables are used to transmit electrical information over long distances. In the future, the amount of data to be transmitted will continue to increase as both the number of people using this technology due to the speed advantages, and the average amount of data per person will increase. The simulation of signal propagation in optical fibers plays an important role in the research and development of optical transmission systems. The interaction of nonlinear and linear effects that occur during propagation in an optical fiber can lead to signal degradation. Therefore, simulations can be helpful to evaluate the influence of nonlinear effects. In general, nonlinear signal propagation in single-mode fibers can be described by the Nonlinear Schrödinger equation (NLSE), which can only be solved analytically for a few cases [5]. Thus, numerical schemes are needed to simulate nonlinear signal propagation in single mode fibers. The most common approach to simulate nonlinear signal propagation is the solution of NLSE with a Split-Step-Fourier-Method (SSFM). However, single-mode fibers are close to reaching their capacity limits, whereas the traffic demand is still growing. Therefore, new approaches need to be investigated. A promising approach to solving this challenge is to use the still untapped spatial dimension. Space-division multiplexing (SDM) has attracted a lot of attention in the last years, both in industry and academia. One way to realize an SDM system is the use of multimode fibers (MMF), where all modes capable of propagation are used as channels for individual signals, referred to as mode-division multiplexing [133].

To investigate the effects of SDM it is necessary to simulate wave propagation in optical fibers. However, simulating the propagation of light in an optical fiber is quite a challenge, as fused silica is a nonlinear medium [5]. The nonlinear signal propagation can be described by coupled partial differential equations for which a closed-form solution only exists in very few special cases. Hence, numerical methods are required to approximate solutions. The investigation of the influence of nonlinear effects in data transmission, is a challenge even for a single propagation mode, because long signal sequences have to be simulated. Therefore, GPU-accelerators are an interesting architecture to speed up simulations [6, 70, 129]. The numerical effort rises sharply when optical fibers with a core diameter $\geq 50\,\mu\text{m}$ are the target of interest: In those fibers several tens/dozens or even more than 100 spatial modes can be used as spatial channels. However, the restricted amount of memory limits the approach to accelerate the simulation of the nonlinear signal propagation [157].

As a result, publications considering numerically the nonlinear signal propagation in MMFs are currently mostly limited to just a few modes if only a single GPU is used. Therefore, in this chapter we introduce the possibility to distribute the simulation of a single transmission scenario to multiple nodes equipped with GPU accelerators or to single nodes with multiple GPUs.

Instead of utilizing split-step Fourier methods as pursued by, e.g., [144], we use the fourth-order Runge-Kutta in the Interaction Picture (RK4IP) method. This method has the potential to reduce the numerical error while simultaneously allowing an increased step size and is thus favorable to reduce the computation time of the simulation [33].

3.2 Modeling of nonlinear signal propagation in multimode fibers

Since this chapter focuses on the improvement of the implementation and the model and methodology should not be changed, we refer to [15, 28, 100] for the derivation of the physical equations and start directly with the resulting PDE. The nonlinear signal propagation in multimode fibers can be described by the nonlinear Schrödinger [131] or the Manakov equation [107, 116] for multimode fibers:

$$\begin{aligned}\frac{\partial A_{\mathfrak{a}}(z,t)}{\partial z} = &\overbrace{\left(-\frac{\alpha}{2} + i\sum_{n=0}^{N_T}\left(\frac{i^n}{n!}\,\beta_{n,\mathfrak{a}}\,\frac{\partial^n}{\partial t^n}\right)\right)}^{\hat{L}} A_{\mathfrak{a}}(z,t) \\ &\underbrace{+\, i\gamma\left(\kappa_{\mathfrak{a}\mathfrak{a}}|A_{\mathfrak{a}}(z,t)|^2 + \sum_{\substack{\mathfrak{b}=0\\ \mathfrak{b}\neq\mathfrak{a}}}^{N_G}\kappa_{\mathfrak{a}\mathfrak{b}}|A_{\mathfrak{b}}(z,t)|^2\right) A_{\mathfrak{a}}(z,t)}_{\hat{N}(A_{\mathfrak{a}}(z,t))}\end{aligned} \tag{3.1}$$

Here, $A(z,t)$ denotes the field envelopes of the mode groups $\mathfrak{a}$ and $\mathfrak{b}$, N_G is the number of groups, α is the attenuation coefficient, the Taylor coefficients of the propagation constants are given by β_n and N_T is the number of Taylor coefficients. Within the nonlinear part $\hat{N}$, the parameter γ is associated with the nonlinear refractive index change which is due to the Kerr-effect. The intramodal nonlinear coupling coefficient is specified as $\kappa_{\mathfrak{a}\mathfrak{a}}$, whereas the intermodal interaction is weighted with $\kappa_{\mathfrak{a}\mathfrak{b}}$. In the context of this thesis we restrict ourselves to constant coupling coefficients κ. For a more detailed description see, e.g., [15]. Equation (3.1) is to be solved for all z in a given interval $[0, L]$ where L denotes the length of the fiber and for all 'local time' $t \in \mathbb{R}$. The 'local time' t is the retarded time traveling at the envelope group velocity. It is considered together with the boundary condition $A(0,t) = a_0(t)$ for all $t \in \mathbb{R}$, where a_0 is a 'smooth' complex valued function [5].

3.3 Numerical approximation

Since the analytical solution can only be calculated for a few special cases, numerical methods are required for the evaluation of equation (3.1). To approximate the solution of equation (3.1), pseudo-spectral methods like the split-step Fourier method (SSFM) [5] or the fourth-order Runge-Kutta in the Interaction Picture (RK4IP) method [76] can be used. In this thesis, we only discuss the RK4IP method. We have performed a comparison of the two methods in [33].

3.3.1 The fourth-order Runge-Kutta in the interaction picture method

To apply pseudospectral methods to solve equation (3.1), the problem is typically separated into a linear and a nonlinear part

$$\frac{\partial A}{\partial z} = \hat{L}A + \hat{N}(A) \tag{3.2}$$

where the linear operator $\hat{L}$ covers the attenuation and the dispersive terms of equation (3.1), and the nonlinear operator $\hat{N}$ governs the nonlinear contributions. For better readability we simply write A for the field envelopes, but keep in mind that this is actually a function $A(z,t)$ depending on the location variable z and the time variable t. These parts are typically solved independently of each other. In the case of the SSFM, the linear part is solved in the frequency domain, whereas the nonlinear operator is solved in the time domain. Thus, interaction between the linear and nonlinear part in equation (3.2) is partially neglected. This leads to a splitting error, given by the Baker-Hausdorf formula [164], limiting the numerical accuracy. However, our experiments [33] indicate that the splitting error not dominates significantly, so that the RK4IP method can achieve better accuracy than the SSFM.

Applying the RK4IP method, equation (3.1) is transformed into the 'Interaction Picture' to decouple the linear and nonlinear operator. The field envelopes are represented by

$$A_I = \exp\left(-(z-z')\hat{L}\right)A \tag{3.3}$$

where z' is the separation distance between the interaction and normal pictures. The differentiated form of equation (3.3)

$$
\begin{aligned}
\frac{\partial A_I}{\partial z} &= \exp\left(-(z-z')\hat{L}\right) \underbrace{\frac{\partial A}{\partial z}}_{=\hat{L}A+\hat{N}(A)} - \exp\left(-(z-z')\hat{L}\right)\hat{L}A \\
&= \exp\left(-(z-z')\hat{L}\right)\left(\hat{L}A+\hat{N}(A)\right) - \exp\left(-(z-z')\hat{L}\right)\hat{L}A \\
&= \exp\left(-(z-z')\hat{L}\right)\hat{N}(A) \\
&= \exp\left(-(z-z')\hat{L}\right)\hat{N}\left(\underbrace{\exp\left((z-z')\hat{L}\right)\exp\left(-(z-z')\hat{L}\right)}_{=1}A\right) \\
&= \exp\left(-(z-z')\hat{L}\right)\hat{N}\left(\exp\left((z-z')\hat{L}\right)A_I\right) := F(z,A_I), \qquad (3.4)
\end{aligned}
$$

where $F(z,u)$ is the nonlinear operator in the interaction picture. Here we get the first line if we derive equation (3.3) using the chain and the product rule. From line one to line two we use equation (3.2). We combine all terms of the second line and get the third line. To transfer the field envelopes into the interaction picture, we perform a identity multiplication in the fourth line. Afterwards, we use the definition of A_I and get the desired PDE in the interaction picture. This can now be solved using explicit schemes like the fourth-order Runge-Kutta method.

In order to solve the PDE, the domain must be discretized. The integration interval $[0,L]$ is first divided into K subintervals. The spatial grid points are denoted z_k, $k \in \{0,\ldots,K\}$ where $0 = z_0 < z_1 < \ldots < z_{K-1} < z_K = L$. For convenience we assume a constant grid spacing $h = L/K$. Furthermore, $A_{I,n} = A_I(z_n,t)$ and $A_n = A(z_n,t)$ are used for better readability. The trick now is to set $z'_k = z_k + \frac{h}{2}$, which leads to several eliminations. Now we solve equation (3.4) using the Runge-Kutta method with:

$$
\begin{aligned}
k_1 =& hF(z_n, A_{I,n}) \\
k_2 =& hF\left(z_n + \frac{h}{2}, A_{I,n} + \frac{k_1}{2}\right) \\
k_3 =& hF\left(z_n + \frac{h}{2}, A_{I,n} + \frac{k_2}{2}\right) \\
k_4 =& hF\left(z_n + h, A_{I,n} + k_3\right) \\
A_{I,n+1} =& A_{I,n} + \left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\right)
\end{aligned}
$$

Substituting equation (3.4) yields

$$\begin{aligned}
k_1 =&hF(z_n, A_{I,n}) = h\exp\left(\frac{h}{2}\hat{L}\right)\hat{N}\left(\exp\left(-\frac{h}{2}\hat{L}\right)A_{I.n}\right)\\
k_2 =&hF\left(z_n+\frac{h}{2}, A_{I,n}+\frac{k_1}{2}\right) = h\exp\left(-(0)\hat{L}\right)\hat{N}\left(\exp\left((0)\hat{L}\right)(A_{I,n}+\frac{k_1}{2})\right)\\
=&h\hat{N}\left(A_{I,n}+\frac{k_1}{2}\right)\\
k_3 =&hF\left(z_n+\frac{h}{2}, A_{I,n}+\frac{k_2}{2}\right) = h\exp\left(-(0)\hat{L}\right)\hat{N}\left(\exp\left((0)\hat{L}\right)(A_{I,n}+\frac{k_2}{2})\right)\\
=&h\hat{N}\left(A_{I,n}+\frac{k_2}{2}\right)\\
k_4 =&hF\left(z_n+h, A_{I,n}+k_3\right) = h\exp\left(-\frac{h}{2}\hat{L}\right)\hat{N}\left(\exp\left(\frac{h}{2}\hat{L}\right)(A_{I,n}+k_3)\right).
\end{aligned}$$

However, we are looking for a solution of equation (3.2). With equation (3.3) we lead the Runge-Kutta method back to A. To make the calculations more efficient, we use equation (3.3) in the computation of k_1 and set $k_4' = \exp(\frac{h}{2}\hat{L})k_4$. All in all we get the following set of equations:

$$A_{I,n} = \exp\left(\frac{h}{2}\hat{L}\right)\cdot A(z_n,t) \tag{3.5a}$$

$$k_1 = h\exp\left(\frac{h}{2}\hat{L}\right)\hat{N}\left(A_n\right) \tag{3.5b}$$

$$k_2 = h\hat{N}\left(A_{I,n}+\frac{k_1}{2}\right) \tag{3.5c}$$

$$k_3 = h\hat{N}\left(A_{I,n}+\frac{k_2}{2}\right) \tag{3.5d}$$

$$k_4' = h\hat{N}\left(\exp\left(\frac{h}{2}\hat{L}\right)(A_{I,n}+k_3)\right) \tag{3.5e}$$

$$A(z_{n+1},t) = \exp\left(\frac{h}{2}\hat{L}\right)\cdot\left(A_{I,n}+\frac{k_1}{6}+\frac{k_2}{3}+\frac{k_3}{3}\right)+\frac{k_4'}{6} \tag{3.5f}$$

The equations result in exactly one iteration of the RK4IP algorithm. Before we present it we discuss the execution of the linear operator $\hat{L}$ and the nonlinear operator $\hat{N}(A_n)$. The linear operator $\hat{L}$ in equation (3.1) can be applied straightforward after transforming it into the frequency domain [5]:

$$-\frac{\alpha}{2}+i\sum_{n=0}^{N_T}\left(\frac{i^n}{n!}\beta_{n,\mathfrak{a}}\frac{\partial^n}{\partial t^n}\right)\circ\!\!-\!\!\!-\; -\frac{\alpha}{2}+i\sum_{n=0}^{N_T}\left(\frac{i^n}{n!}\beta_{n,\mathfrak{a}}\left(-i\omega\right)^n\right) \tag{3.6}$$

Here, the attenuation is included in the real part, whereas the dispersive terms are covered by the imaginary part. The symbol '$\circ\!\!-\!\!\!-$ ' describes the Fourier transformation from time to frequency domain. Therefore, the equations (3.5a)–(3.5f) require eight Fourier transformations (FFTs), two in each equation (3.5a), (3.5b), (3.5e), and (3.5f). Without using the trick of setting $z' = z + \frac{h}{2}$, this approach would require 16 FFTs per step because multiple $\exp((z - z')\hat{L})$ terms would not become identity. In contrast to the linear operator, the nonlinear operator can be calculated in the time domain without problems. Here, everything except the two matrix-matrix multiplications $\kappa|A(z,t)|^2$, are element-wise operations.

The method is globally fourth-order accurate [12]. The precision can be further increased by implementing a local error method [19, 69, 168], and further be combined with an adaptive step size control [19, 69]. As a result we get algorithm 3.1, where the linear operator is described in algorithm 3.2 and the nonlinear operator in algorithm 3.3.

Algorithm 3.1 (RK4IP-algorithm).

```
1: Input: A, kappa, h
2: Output: A
3: lin_op = calc_lin_op()                                          ▷ eq. (3.6)
4: for $z = 0;\ z < L;\ z = z + h$ do
5:     k1 = nonlinear_operator(A, kappa)          ▷ Nonlinear operator in eq. (3.5b)
6:     for mode=0; mode < number_of_modes; mode++ do
7:         A(mode) = FFT(A(mode))
8:         A(mode) = lin_op_exe(A(mode), lin_op, h)                ▷ eq. (3.5a)
9:         A(mode) = IFFT(A(mode))
10:    end for
11:    for mode=0; mode < number_of_modes; mode++ do
12:        k1(mode) = FFT(k1(mode))
13:        k1 = lin_op_exe(k1(mode), lin_op, h)                    ▷ eq. (3.5b)
14:        k1(mode) = IFFT(k1(mode))
15:    end for
16:    k2 = h · nonlinear_operator(A + 0.5 · k1, kappa)            ▷ eq. (3.5c)
17:    k3 = h · nonlinear_operator(A + 0.5 · k2, kappa)            ▷ eq. (3.5d)
18:    for mode=0; mode < number_of_modes; mode++ do
19:        k4(mode) = FFT(A(mode) + k3(mode))
20:        k4(mode) = lin_op_exe(k4(mode), lin_op, h)              ▷ eq. (3.5e)
21:        k4(mode) = IFFT(k4(mode))
22:    end for
23:    k4 = h · nonlinear_operator(k4,kappa)                       ▷ eq. (3.5e)
24:    for mode=0; mode < number_of_modes; mode++ do
25:        sum_tmp = calc_weighted_sum(A, k1, k2, k3)
26:        sum_tmp(mode) = FFT(sum_tmp(mode))
27:        sum_tmp(mode) = lin_op_exe(sum_tmp(mode), lin_op, h)
28:        sum_tmp(mode) = IFFT(sum_tmp(mode))
29:    end forA = sum_tmp + (1/6) · k4                             ▷ eq. (3.5f)
30: end for
```

Algorithm 3.2 (lin_op_exe).

```
1: Input: A, lin_op, h
2: Output: L
3: for $k = 0;\ k < length(A);\ k{+}{+}$ do
4:     L(k) = cexp(lin_op(k) · 0.5 · h)
5:     L(k) = cmul(A(k),L(k))
6: end for
```

Algorithm 3.3 (nonlinear operator).

```
1: Input: A, kappa
2: Output: N
3: for $k = 0;\ k < num\_row(A);\ k{+}{+}$ do
4:     for $l = 0;\ l < num\_col(A);\ l{+}{+}$
   do
5:         sqr_abs(k,l) = cabs(A(k,l))
6:         sqr_abs(k,l) *= sqr_abs(k,l)
7:     end for
8: end for
9: tmp = cmatMul(kappa,sqr_abs)
10: for $k = 0;\ k < num\_row(A);\ k{+}{+}$ do
11:     for $l = 0;\ l < num\_col(A);\ l{+}{+}$
   do
12:         A(k,l) = cmul(tmp(k,l), A(k,l))
13:     end for
14: end for
```

3.4 Numerical simulation

For the numerical simulation of the nonlinear signal propagation in optical fibers the signals can be stored as sampled data. These data can be represented by a matrix, where either rows or columns are aligned linear in the memory (see figure 3.1). For the investigation of an SDM transmission it is typical to consider much more samples per mode than number of modes, so the matrix is typically in the size of 30×2^{20}. Since all samples of a single mode corresponding to a single row, are required for the Fourier transform (FFT), it is advantageous to align the rows rather than the columns linearly in memory. In the following we call this matrix *data matrix.* The data matrix has the dimension $2 \cdot M \times N$, where M is the number of modes. The number of polarization planes is taken into account by the factor 2, and N is the number of samples. FFTs are required before and after every computation of the linear operator $\hat{L}$, cf. equation (3.5) and algorithm 3.1. It is important to note that the nonlinear interaction between the modes at a given point in time is represented by a matrix-matrix multiplication in the time domain, where both matrices are dense. More precisely, the nonlinear interaction is given by an interaction matrix, containing the nonlinear coupling coefficients κ, multiplied by the element-wise square of the absolute value of the data matrix, followed by an element-wise multiplication with the data matrix (see equation (3.1) and algorithm 3.3). The interaction matrix is a square matrix of the dimension $2 \cdot M \times 2 \cdot M$, with the nonlinear coupling coefficients $\kappa_{\mathfrak{aa}}$ on the main diagonal, and where the nonlinear coupling coefficients $\kappa_{\mathfrak{ab}}$ are the non-diagonal elements. Remember that the coupling coefficients are constant and therefore the interaction matrix is constant. Therefore, additional column-wise access to the memory is required (see figure 3.1).

All other operations of the algorithm are elementary additions, multiplications and evaluations of trigonometric functions. For each sample, these operations are independent of the other samples. Generally all operations, except the matrix multiplication, are straightforward to parallelize. The linear operator $\hat{L}$ in equation (3.1) can be applied straightforward after transforming it into the frequency domain. Here, the attenuation is included in the real part, whereas the dispersive terms are covered by the imaginary part. Precomputing and storing the operator for all modes is possible and requires the same amount of memory as for the signals. This is represented in line 3 in algorithm 3.1.

3.4.1 CPU implementation

The CPU implementation uses FFTW [57] for the fast Fourier transforms, and is parallelized with OpenMP [127]. The CPU implementation using OpenMP is chosen because it is an easy to parallelize approach which is adequate for a fair node to node comparison. Furthermore, this comparison shows that the GPU implementation offers strong advantages so that the step from one to multiple nodes is only implemented for the GPU implementation. As described in the previous section, all operations except matrix-matrix

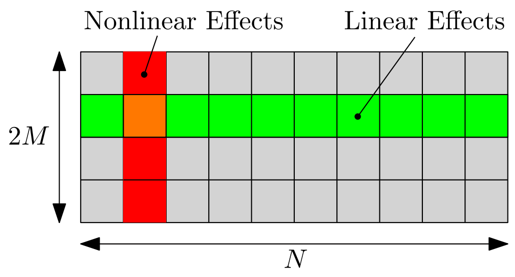


Figure 3.1: Representation of the data matrix, where M is the number of modes and N is the number of samples. The nonlinear effects occur during the matrix-matrix multiplication with the interaction matrix from left.

multiplication can be easily parallelized. Therefore, all computations, except the matrix multiplication, are parallelized with static for-loops, oblivious to the number of cores. The matrix multiplication for the nonlinear interaction is implemented as a sequential loop over all spatial and polarization channels, i.e., the $2 \cdot M$ rows of the square interaction matrix. The inner loop, which represents the inner products between the row of the interaction matrix and each column of the element-wise squared data matrix, is parallelized over the columns. This loop also contains the element-wise multiplication with the data matrix. These loops therefore replace lines 9-13 in algorithm 3.3. That is, all entries of one row of the output matrix are calculated in parallel. Recall that the element-wise squared data matrix (cf. lines 5 and 6 in algorithm 3.3) is aligned linear by rows just like the data matrix, whereas column access is required for the matrix multiplication, which increases cache misses. For the same reason, it is not possible to vectorize the matrix multiplication. This means that either the best data structure for the FFT or the matrix multiplication must be chosen. In [70] and [131], it is shown that the FFT is the bottleneck for a small number of channels. We follow these findings and optimize the data structure for the FFT, i.e., we decide to align the data matrix row-wise in memory.

3.4.2 CPU/GPU hybrid implementation

Since the algorithm requires eight Fourier transforms, which are a very complex tasks, outsourcing only these tasks to the GPU might be a good idea. Furthermore, this CPU/GPU hybrid variant serves as a comparison between the full CPU and the full GPU variant. Therefore, we replace the FFTW library by the cuFFTW library. The latter provides an FFTW-compatible interface to the 'NVIDIA CUDA Fast Fourier Transform library' (cuFFT) [125] library and can be used as a drop-in replacement for the FFTW library. With this implementation, all FFTs are performed on the GPU, since cuFFT is a library to compute fast Fourier transforms on the GPU. However, the data is automatically copied between the host and the device memory (and vice versa) before and after each fast Fourier transform, as all other operations are executed by the CPU. This might introduce a performance drawback as discussed in section 3.5.

3.4.3 GPU implementation

We recall that all operations except for matrix multiplication are easy to parallelize. Therefore, the GPU implementation for the linear operations is straightforward [67], using one thread for each sample. All memory accesses are contiguous, and thus optimal for the parallel memory architecture of the device. We employ cuFFT [125] for the fast Fourier transforms.

As with the CPU/GPU hybrid implementation, it is necessary to copy all data from host to device memory and vice versa before and after computations on the GPU. As the PCIe bus compared to the bandwidth for on board DRAM access is slow, these additional data transfers can easily diminish all speedup obtained from computing on the device. This problem can be mitigated by using asynchronous copies, that overlap copy and computation for independent data. In CUDA, this is accomplished by so-called streams, defined as a sequence of operations that execute on the device in the order in which they are issued by the host code (see section 2.2.6). It is important to note that operations in different streams can be nested and thus executed simultaneously.

In our case we choose one stream for each channel. When the portion of the data corresponding to the first channel has been copied to the GPU, all 'linear' computations for the first channel can start. During the calculations for the first channel the copying for the next channel is performed, resulting in overlapping of computation and data movement. The same principle can be applied to the following channels, so that all data movements, except for the first channel, can be hidden by computations. The same procedure is possible after the last iteration, because the data must be available again on the host for postprocessing. Once all computations for one channel are completed, the copy of the data for this channel to the host can start, while the computations for the other channels are still running.

Assigning one stream for each channel enables better latency hiding in the linear computation phase: This phase is strongly limited by memory bandwidth. During two consecutive computations for the same sample, the second computation only has to wait until the first sample is completed for the related channel instead of all channels. Because of this, it is possible to overlap the linear phase of one channel with the FFTs of the others. In summary, we obtain a perfectly asynchronous algorithm, which is only synchronous in the matrix-matrix multiplications, i.e., in the nonlinear coupling.

In this matrix-matrix multiplication, each element of the output matrix is an inner product of a row of the interaction matrix and a column of the data matrix. A naive approach is that each thread is responsible for calculating one element of the output matrix. However, this results in $2M$ (which is the number of channels) memory accesses to the elements from a row of the interaction matrix and $2M$ accesses to the entries from a column of the data matrix. Altogether, there are $(2 \times (2M)^2 \times N)$ read accesses, and data reuse is only implicit through the cache hierarchy on the device. This can also be seen in figure 3.2, where this approach is illustrated. A better strategy is to use so-called

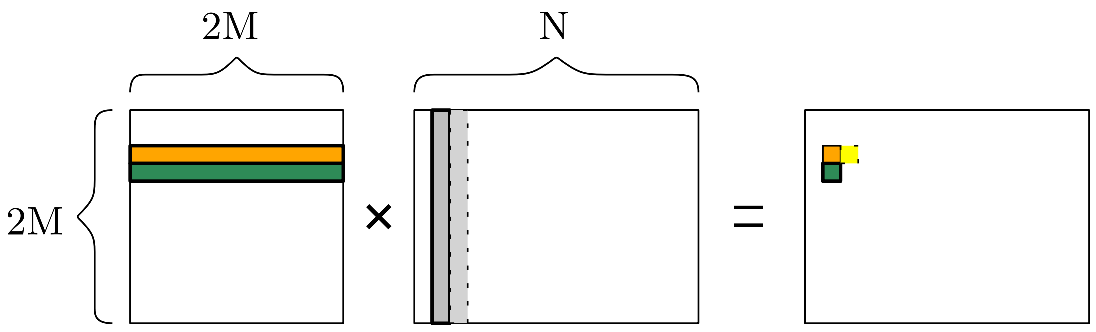


Figure 3.2: Block approach for a matrix-matrix multiplication.

CUDA shared memory, a small but fast scratchpad memory for each multiprocessor, similar to a programmable cache (see section 2.2.5). This means that the entire matrices are divided into equal subsets, so that a tiled pattern arises: All threads responsible for a tile of the output matrix collaboratively load subsets of the input matrices into shared memory. Then, they individually use these elements in their inner product calculation. One possible choice for these subsets would be, e.g., 32×32 matrix blocks. By loading each global memory value into shared memory so that it can be used multiple times, we reduce the number of accesses to the global memory. This can also be seen in figure 3.3, where this approach is illustrated. First the two grey blocks are loaded and the matrix-matrix multiplication is executed for them. Then the next block in the row or column is loaded and the matrix-matrix multiplication is performed for them. The results are summed to the output matrix.

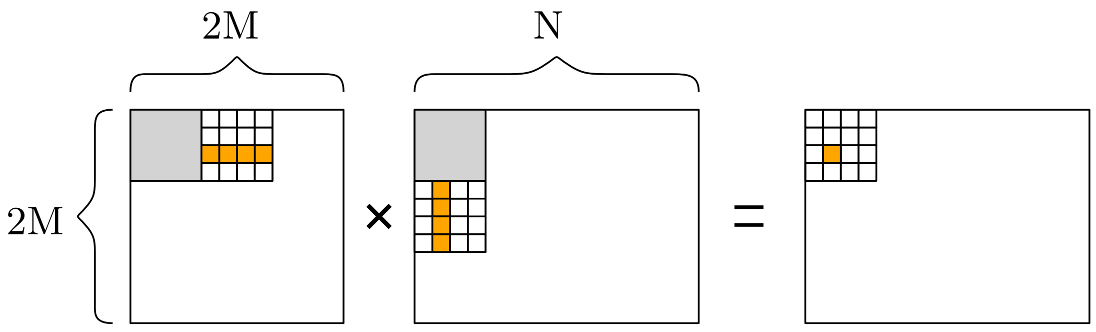


Figure 3.3: Naive approach for a matrix-matrix multiplication.

In our case, the interaction matrix is constant and does not change in the complete algorithm. It also fits into so-called constant memory, which is another specialized memory region optimized for read-only access. We thus benefit in two ways, since only the data matrix needs to be stored in shared memory, reducing shared memory pressure, because more tiles of the data matrix can be stored in shared memory simultaneously.

As a further optimization, we exploit that the data matrix has a large number of columns (equal to the number of samples), but only a small number of rows (equal to the number of channels). Hence, a whole column is only part of a small number of matrix tiles. This is of advantage, because it is necessary to synchronize the threads before and after loading each tile into shared memory. This becomes clear when considering figure 3.3 again. The two gray blocks are loaded into shared memory first. However, before the matrix-matrix multiplication can be performed for these blocks, it must be ensured that all threads of these blocks have loaded the data. At this point we remember again that only for a warp can be guaranteed to execute the instructions simultaneously. We finally note that loading the tiles of the data matrix into the shared memory, and the read operations from the shared memory, are possible without any bank conflicts (c.f. section 2.2.5). The same applies to the interaction matrix.

Additionally, all CUDA kernels are optimized by using standard techniques as described in [82].

Precomputing vs. less memory requirement The problem size for real application setups quickly becomes so large that it reaches the capacity of the GPU memory. How big such problems become is described in the next section as an example. Before we show how to solve the problem on multiple GPUs in parallel, which is essential for problems of such practically relevant sizes, we will now describe why it is possible to avoid precomputations without losing performance. In other words, we discuss how to reduce the amount of required global GPU memory. The linear operator $\hat{L}$ in equation (3.1) can be applied straightforward after transforming it into the frequency domain [5]:

$$-\frac{\alpha}{2} + i \sum_{n=0}^{N_T} \left(\frac{i^n}{n!} \beta_{n,\mathfrak{a}} \frac{\partial^n}{\partial t^n} \right) \circ\!\!-\; -\frac{\alpha}{2} + i \sum_{n=0}^{N_T} \left(\frac{i^n}{n!} \beta_{n,\mathfrak{a}} \left(-i\omega\right)^n \right)$$

Here, the attenuation is included in the real part, whereas the dispersive terms are covered by the imaginary part. Precomputing and storing the operator for all modes requires the same amount of memory as for the signals. However, the cuFFT library computes an unnormalized Fourier transform. In other words, applying the forward and the backward transform leads to the multiplication of the signals by the number of samples N_s. Hence, the result needs to be normalized with $1/N_s$. Since the attenuation coefficient α is often assumed to be frequency independent for the considered frequency range [5], α can be taken into account by simply extending the normalization by $\exp(-\alpha/2 \cdot h)$, where h is the step size of the linear sub-step (as described in section 3.3.1). Thereby, the precomputation of the real part can then be skipped, also causing fewer memory accesses. Going further, we do not precompute the linear operator at all. Only the sampled frequency vector containing ω and the factorials $n!$ are precomputed. In consequence additional calculations need to be performed on the GPU. Anyway, applying the imaginary part $\Im\{\hat{L}\}$ is limited by the memory bandwidth and the additional computations are scheduled while waiting for memory access. In this case the hiding of latencies is used to avoid precomputations (line 3 in algorithm 3.1) and to save memory space. This means that the precomputations vanish in algorithm 3.1, and instead are performed in the loop of algorithm 3.2. As the latencies are exploited with computations, the workload of the GPU automatically increases and the performance comes closer to peak performance.

Also for this approach the constant memory can be used to store the beta values as well as the fatcorials and thus reduce latencies.

3.4.4 Multi-GPU implementation

As already mentioned, the signals can be represented by a matrix of sampled data with the dimension $2M \times N$, where M is the number of modes, the number of polarization planes is taken into account by the factor 2, and N is the number of samples. For the investigation of Kerr-based nonlinear effects for practically-relevant simulations, however, these data quickly become very large, e.g., each symbol needs to be represented by an appropriate number of samples to simulate a sufficiently large frequency spectrum. E.g. 256 samples per symbol, denoted as N_{sps}, were used in [32] to simulate a spectral range of 8.192 THz. In the referenced simulation, $M = 15$ spatial modes and $N_\mathrm{s} = 2^{14}$ symbols per spatial and per polarization mode were considered. With $N = N_s \cdot N_\mathrm{sps}$, this results in a complex valued dense matrix of size 30×2^{22}, which requires 1920 MiB of storage. When further increasing the number of spatial modes M, the matrix containing the sampled signal might still fit into the GPU-memory, but not all intermediate results do any longer. We therefore propose to split the $2M$ polarization and spatial modes to K processes. Since $N \gg 2M$, this approach has several advantages over splitting N contiguous samples of a unique spatial or polarization mode to different processes, as discussed in the next section. Based on this splitting we also present a multi-GPU algorithm.

Splitting the numerical problem In principle there are two approaches how to parallelize the problem. Roughly speaking, the data matrix can be split either horizontally or vertically. The second approach requires a parallelization of the Fourier transformation. In [144, 169] the split-step Fourier method is parallelized by using distributed fast Fourier transform implementations. However, this requires a lot of communication between the involved compute nodes. We choose the other approach where instead of letting several processes take part in the calculation of a spatial or polarization mode, only whole modes are distributed to the different processes. Here, each process is associated with one GPU, but the process itself can still involve multiple threads. Thus, the N samples of a single signal are only required and processed by one unique process. As proposed, the channels are equally distributed to K processes. With this, each process computes $2M/K$ channels as illustrated in figure 3.4.

The matrix representing the sampled signal is stored row-major and the rows are aligned linearly in the memory. Therefore, the memory alignment is optimized for the fast Fourier transforms (FFT), as discussed in section 3.4. The computation of the linear step $\hat{L}$ can be executed fully parallel by each process independently. Only for the calculation of the nonlinear step $\hat{N}$ information from the other processes is required, namely the squared absolute values of the envelopes A of all modes that are not locally available.

The SSFM requires the computation of $|A|^2$ once at the position z for the first iteration

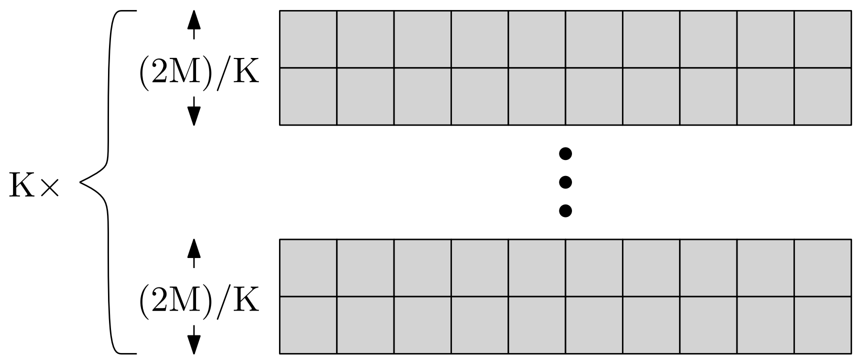


Figure 3.4: Signal matrix split to multiple processes.

and at the position $z + h$ for every following iteration. Using the SSFM-RK4, the values $|A|^2$ are required to calculate k_1, k_2, k_3, and k_4, which is the same for the RK4IP method (c.f. equation (3.5b)–(3.5e)). The squared absolute values $|A|^2$ can be stored real-valued. Therefore, in every iteration or rather the calculation of k_n, $(2M - 2M/K) \cdot N$ real valued numbers have to be provided by the other processes and each process has to share its $(2M/K) \cdot N$ computed values. The squared absolute values $|A|^2$ are exchanged via MPI or via the NVIDIA Collective Communications Library (NCCL). Due to the large signal matrices, one has to expect quite large messages even if communication is kept minimal with our splitting approach. For the previous example with matrix size 30×2^{22}, sharing all squared absolute values would result in a total message size over all processors per iteration of 960 MiB (before line 9 in algorithm 3.3).

In the following we apply our modifications to the RK4IP method again. Note, however, that the presented approach can be applied in an identical way to the general SSFM and the SSFM-RK4.

GPU communication As described above, physically relevant problems are often too large and exceed the memory of a GPU. However, they do not become arbitrarily large either, that is why only a few GPUs are needed to solve such problems, not hundreds, making small clusters the target platform. Nevertheless there should be no restriction that all GPUs are on the same board, so that MPI communication is necessary (see section 2.3). Since the messages are very large, we use NCCL in addition to our own communication with MPI. NCCL provides topology-aware collective communication primitives and is well suited for handling large messages.

MPI-implementation One option to realize the communication between the involved GPUs is to use the the Message Passing Interface (MPI) [108]. Using MPI has the advantage, that the GPUs do not necessarily have to be placed in the same compute node. Here, one MPI process per GPU is used. With the availability of CUDA-aware MPI [85] implementations, the programmer does not have to stage the data in the host memory, as the GPU buffers can be directly passed to MPI.

A naive approach to realize the communication via MPI is the use of collective operations like `MPI_Bcast` or `MPI_Allgather`. However, these rely on blocking communication and CUDA-aware implementations that, supporting non-blocking collectives, are still under development [17, 18]. Using non-blocking communication instead has the advantage to overlap communication and process the data. Overlapping communication and computations are essential to hide communication costs and to obtain good scalability. We therefore decided to explicitly exchange data via asynchronous, and therefore non-blocking send and receive operations, namely `MPI_Isend` and `MPI_Irecv`. The program sequence is described in listing 3.1.

```
void send_sqrabs(const int rank,
  const int size, const REAL *sqrabs,..,
  const int num_elem, MPI_Request *send_req) {

  for(int rk=0;rk<rank;rk++)
    MPI_Isend(&sqrabs[..],num_elem,
      MPI_DOUBLE,..,MPI_COMM_WORLD,
      &send_req[rk]);

  for(int rk=rank+1;rk<size;rk++)
    MPI_Isend(&sqrabs[..],num_elem,
      MPI_DOUBLE,..,MPI_COMM_WORLD,
      &send_req[rk-1]);
}

void recv_sqrabs(const int rank,
  const int size, REAL *sqrabs,..,
  const int num_elem, MPI_Request *recv_req) {

  for(int rk=0;rk<rank;rk++)
    MPI_Irecv(&sqrabs[..],num_elem,
      MPI_DOUBLE,..,MPI_COMM_WORLD,
      &recv_req[rk]);

  for(int rk=rank+1;rk<size;rk++)
    MPI_Irecv(&sqrabs[..],num_elem,
      MPI_DOUBLE,..,MPI_COMM_WORLD,
      &recv_req[rk-1]);
}

calc_squareabs(..);
recv_sqrabs(..);
send_sqrabs(..);
calc_nonlinear_own(..);

all_done = 0;
while(all_done < size-1) {
  MPI_Waitany(size-1, recv_req, &rk_idx,..);
  calc_nonlinear_others(..,rk_idx,..);
40   all_done++;
41 }
42
43 apply_nonlinear_all(..);
```

Listing 3.1: Basic program flow to compute $\hat{N}$ incorporating MPI.

This program code replaces the lines 9-14 in algorithm 3.3 The function `send_sqrabs` calls the `MPI_Isend` function, whereas in line 5-8 the process sends its own data to processes with lower rank numbers and in line 10-13 to all processes with higher rank numbers. In lines 16-19 `recv_sqrabs` is defined, which is similar to the send function and calls the `MPI_Irecv` functions accordingly. After the computation of $|A|^2$ for the $(2M)/K$ modes persisting on the GPU (line 31 in listing 3.1), we initialize the data exchange operations. The values are send via `MPI_Isend` in the `send_sqrabs()` function (line 32 in listing 3.1), and matching receive `MPI_Irecv` commands are posted in the `recv_sqrabs()` function (line 33 in listing 3.1). As mentioned before, these operations are non-blocking and therefore both commands return immediately, even if the transfers are not finished. Next, the CUDA kernel is launched to calculate the contribution to the nonlinear phase rotation of the modes that are persisting on the GPU (line 34 in listing 3.1), which is non-blocking again. Afterwards in line 38, a blocking operation `MPI_Waitany` is called, to wait until any of the `MPI_Irecv` commands has finished and the contribution of the received $|A|^2$ values to the nonlinear phase rotation is calculated in line 39. If all $|A|^2$ values of the $K-1$ other processes are received, and all contributions are taken into account, the nonlinear phase rotation is finally applied to the modes persisting on the GPU.

This approach scales perfectly if the time needed to receive the next data is shorter than the time for the simultaneously performed computations. In this case, the GPU does not have to wait for the next data, since these are received while the GPU is performing computations. The first work package, i.e. line 34, is always available on the GPU, since this is the calculation of `calc_nonlinear_own()` for which no data needs to be received. However, the execution of `calc_nonlinear_others()` relies on data sent from the other processes. In practice, the possible overlap strongly depends on the simulation set-up, i.e., the number of spatial modes M and samples N, and is limited by the number of involved GPUs K as well as the interconnects between the GPUs.

NCCL-implementation A higher-level approach is to exchange data via the NVIDIA Collective Communications Library (NCCL). NCCL supports multiple GPUs installed in a single node or across multiple nodes. The library provides topology-aware collective communication primitives and features multiple ring formations for high bus utilization. Within NCCL, the collectives are implemented in a single kernel and are therefore associated to a so-called CUDA stream [165]. The NCCL calls return when the operation is enqueued to the specified stream and the collective operation is executed asynchronously. In our implementation, `ncclAllGather` is used to aggregate the data. As depicted in

```
calc_squareabs(..);
ncclAllGather((const void*)&sqrabs[..],
              (void*)sqrabs, num_elem,
              ncclDouble,comm,stream_a);

calc_nonlinear_own(..,stream_b);
cudaStreamSynchronize(stream_b);

for(int rk_idx=0; rk_idx < size-1; rk_idx++)
  calc_nonlinear_others(..,rk_idx,..,stream_a);

apply_nonlinear_all(..,stream_a);
```

Listing 3.2: Basic programm flow to compute $\hat{N}$ incorporating NCCL.

listing 3.2, we use different streams for the kernel launch within `calc_nonlinear_own()` and the remaining kernel calls to enable concurrent execution.

To enable the implementation to utilize multiple nodes, we use NCCL together with MPI. Hence, each GPU is associated with an MPI process as before. A common NCCL communicator spanning all processes, can be initialized as described in [124, Example 2: One Device per Process or Thread].

Further GPU acceleration In addition to the implementation details for the single GPU variant already presented in section 3.4.3, we present further modifications for the multi-GPU version below.

In the preceding single-node implementation, only a single CUDA kernel capturing the nonlinear effects was launched. As shown before, this is now split up into an `calc_nonlinear_own()` kernel, that is responsible for calculating the nonlinear phase rotation of the locally stored modes, and an `calc_nonlinear_others()` kernel, responsible for the calculation for the nonlinear phase rotation induced by the modes not locally available. Thus, $K - 1$ instances of the latter kernel have to be launched. The overall nonlinear phase rotation is stored in an additional array of size $(2M/K) \cdot N$. Both kernels incorporate so-called shared memory, to alleviate the penalty occurring due to column access to the memory as it is described in section 3.4.3 for the single GPU implementation. In contrast to the single-node implementation, applying the nonlinear phase rotation to the locally available modes now only requires row access instead of column access to the memory. Applying the nonlinear phase rotation is performed by an additional kernel, as already indicated in listings 3.1 and 3.2.

In addition, the interaction matrix no longer fits into the constant memory for a large number of modes without using a splitting approach. Storing all κ values, requires a matrix of $2M \times 2M$ elements. Assuming a symmetric matrix, which is the case for linearly polarized (LP) modes [131], it is sufficient to only store the upper triangular matrix, reducing the number of elements to $(2M \cdot 2M)/2 + M$. This is exemplified for the case of $M = 2$ and $K = 2$ in figure 3.5. For a large number of modes, e.g., $M = 120$, still 28920 double precision values of 8 B would needed to be stored in the constant

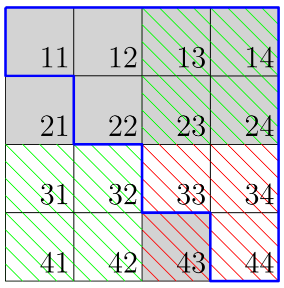


Figure 3.5: Exemplary interaction matrix for $M = 2$ and $K = 2$. The upper triangular matrix is highlighted in blue; All elements required for the calculations on GPU 2 are shaded red and green. By considering the symmetry, the light and dark shaded green elements are identical and only one of the sub-matrices needs to be stored. Furthermore, the light red shaded element can be neglected.

memory, of which only 64 KiB are available. Therefore, this approach does not lead to a sufficient saving. However, only $2M/K \cdot 2M \cdot 2 - (2M/K)^2$ need to be accessed for the calculations. For GPU 2, these are the red and green shaded elements in figure 3.5. The remaining elements of the matrix are only required on the other involved GPU. Taking the symmetry into account again, it is sufficient to save only rows or columns which apply to the modes considered on the certain GPU. With this in mind, the number of elements can be reduced to $2M/K \cdot 2M$. Furthermore, for the κ coefficients describing the nonlinear coupling for the modes persisting in the GPU, it would be sufficient again to only store the upper triangular matrix, as visualized in figure 3.5. However, distributing the matrix via MPI and the necessary index arithmetic is more complicated for this case, and only $(2M/K \cdot 2M/K)/2 - M/K$ additional elements can be saved.

3.5 Numerical results

3.5.1 Performance comparison between CPU and GPU

The benchmarks in this subsection are performed on a workstation with one Intel® Xeon® CPU E5-2620v3 providing 6 physical and, with Hyper-Threading enabled, 12 logical cores. Intel® Turbo Boost is disabled and the CPU frequency is set to 2.4 GHz. The system provides 32 GB DDR4-ECC-RAM and 12 GB GDDR5-ECC-SDRAM and it also contains GPU is a NVIDIA Tesla K40c, that is used for the GPU benchmarks.

In our test case, each spatial mode carries two channels through polarization-division multiplexing. Each signal in a given channel consists of 2^{15} symbols, where each symbol is represented by $N_{\text{sps}} = 32$ samples. Depending on the number of spatial channels, the resulting signal matrix has the dimension $[2, 6, 12, 20, 30] \times 2^{20}$. For the benchmark, 1000 steps are simulated. All calculations are executed with double precision.

Number of	Wall time T in s			$T_{\text{CPU}}/T_{\text{GPU}}$	$T_{\text{CPU}}/T_{\text{HYB}}$
Spatial Modes	CPU	CPU+GPU	GPU		
1	452.5	253.8	18.5	24.5	1.8
3	939.8	829.3	53.8	17.5	1.1
6	2140.5	2070.8	107.6	19.9	1.0
10	6687.3	6559.5	179.6	37.2	1.0
15	14027.6	13689.1	274.8	51.0	1.0

Table 3.1: Runtime and Speedup using 12 cores (chip vs. chip, no MPI).

Table 3.1 lists runtimes and the speedup we obtained. Comparing the runtimes of the OpenMP CPU implementation with the CPU/GPU hybrid implementation, in which only the FFTs are executed on the GPU (denoted as CPU+GPU and the acceleration compared to the CPU implementation as $T_{\text{CPU}}/T_{\text{HYB}}$), it can be found that the implementation can profit from GPU-accelerated FFTs, mainly for a single mode. However, the performance benefit decreases with an increasing number of modes. As previously mentioned, the signal matrix is copied before and after each FFT in this approach, clearly leading to a performance degradation for larger matrices since more data need to be copied. Executing the whole RK4IP algorithm on the GPU eliminates this copy overhead. The speedup for the single-mode simulation is around ~ 25, and slightly decreases for the next two cases, with 3 and 6 modes. With the single-mode simulation, it is not possible to fully utilize the CPU. This becomes clear when we compare the runtime for 1 and 3 spatial modes. The number of polarization and spatial channels, corresponding to the number of unknowns, increases by a factor of three, but the runtime only grows by a factor of two. For more modes, the speedups increase. To explain this, we note first that the CPU scales poorly and unevenly with the number of modes. We consider the runtimes of separate routines for the simulation of 15 spatial modes. The percentage runtimes of the calculations requiring column access and of calculations requiring row access are given in table 3.2. To recap, column-wise access is only required for the calculation of the nonlinear interaction, whereat the squares of the absolute values are precomputed and therefore only require row-wise memory access. Furthermore, the runtime consumed by the FFT are explicitly given in table 3.2. The largest fraction of runtime $\sim 68.8\,\%$

Device	Percentage Wall time		
	FFT	Column Access	Row Access
CPU	12.3%	68.8%	18.9%
GPU	52.3%	12.5%	35.2%

Table 3.2: Percentage runtime for 15 spatial modes.

is consumed by applying the nonlinear interaction. Note that the data matrix is aligned by rows, which leads to cache misses and prevents vectorization by the compiler. Additionally, it is shown in [44] that the scalability of the FFTW is suboptimal as well. Unfortunately, it is only possible to optimize the data structure either for the FFT or for the matrix multiplication as described above. We expect that for an even larger number of modes, optimizing for the latter will be more beneficial.

In contrast, the GPU implementation exhibits very good scalability, i.e., the runtime increases with almost the same factor as the number of unknowns. For further analysis, we examine the runtime consumed by the FFT for the simulation of 15 spatial modes, using the NVIDIA CUDA Profiler. While $\sim 12.3\,\%$ of the runtime are consumed by the FFTW on the CPU, cuFFT is actually responsible for $\sim 52.3\,\%$ of the runtime. Therefore, the measured speedup is not caused by using the cuFFT library. Instead, it can be argued that the speedup is mainly caused by the other computations, which are parallelizable on the GPU in a more efficient way. The biggest improvement is the result of the nonlinear interaction, which accounts for only a percentage of $\sim 12.5\,\%$ in contrast to $\sim 68.8\,\%$ runtime on the CPU. This shows, that it is possible to optimize the matrix multiplication and in particular the FFT on the GPU as well.

All in all the GPU is the much better architecture for this application, because it is possible to optimize each routine of the algorithm. In contrast, programming for the CPU, requires the decision to either optimize for the nonlinear interaction or for the FFT.

3.5.2 Comparison of multi-GPU approaches

To achieve the maximum performance, peer-to-peer access between the GPUs is essential. The benchmark is therefore performed on an AWS EC2-instance of type *p2.8xlarge*. This instance incorporates 4 Tesla K80 accelerators. Each K80 provides a pair of GK210 GPUs, resulting in 8 available GPUs. On this instance type the GPUs are connected via a common PCIe fabric.

The configuration used for the benchmark is given in table 3.3. Incorporating 8 GPUs

M	N_s	N_{sps}	K	M/K
15			1	
30			2	
60	2^{14}	128	4	15
90			6	
120			8	

Table 3.3: Configuration used for the benchmark.

allows to evaluate the nonlinear interaction between 120 spatial modes. This is of interest as it is the number of the potentially usable spatial modes in a fiber with 62.5 µm core diameter [139].

The number of involved processes, or rather GPUs, is scaled up from 1 to 8, to investigate the scaling of the proposed implementations. The number of spatial modes M persisting per GPU is kept constant. In consequence, the total signal matrix occupies up to 7680 MiB, of which 960 MiB are stored per GPU. For every calculation of $\hat{N}$, each GPU needs to share 480 MiB. For the benchmark, 150 steps have been simulated and all calculations are executed with double precision. Recall, that $\hat{N}$ is calculated 4 times per step. This is the same for an SSFM-RK4 implementation, whereas the number to calculate $\hat{N}$ depends on the number of iterations in an SSFM-Agrawal implementation. The initial distribution and the final collection of the sampled signal matrix, as well as the transfer of further necessary parameters and data, is excluded from the benchmark. Results are shown in figure 3.6. Here, the execution times T_K are normalized to the execution time

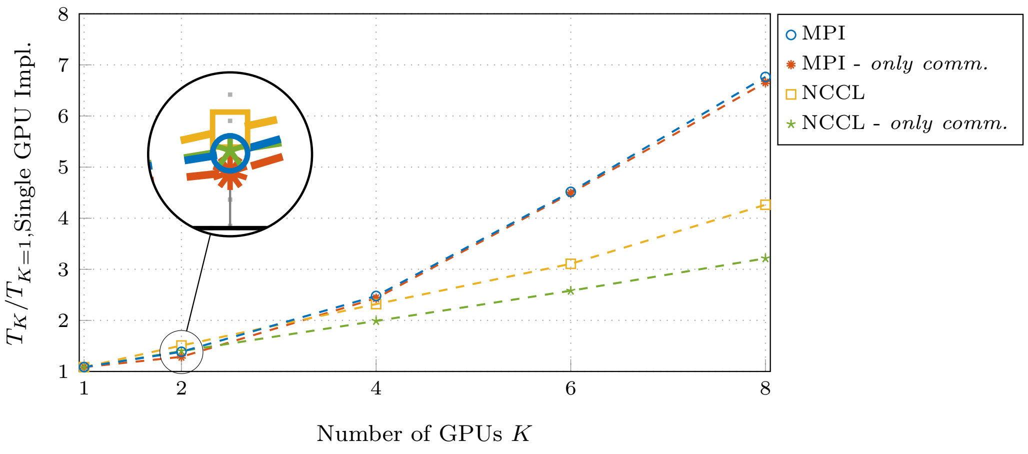


Figure 3.6: Scaling of the implementation.

of our previous single-node, single-GPU implementation [33, 34].

With only a single GPU used, $K = 1$, the relative runtime is > 1. Due to splitting the calculation of $\hat{N}$ into several kernels, the runtime increases by approximately 8.5 %. For $K \leq 4$, the MPI- and NCCL-implementation scale nearly equally. With even more GPUs involved, the execution time of the MPI-implementation rises sharply. Incorporating all 8 GPUs, the MPI-implementation requires 6.76 times the execution time of the single-GPU implementation, whereas the NCCL-implementation scales with a factor of 4.26. To evaluate the reason, the benchmarks are rerun without the additional calculations performed in the `calc_nonlinear_others()`. Therefore, only the amount of communication grows with an increasing K. Here, the MPI-implementation shows nearly identical results, whereas the relative runtime of NCCL-implementation drops. For the MPI-implementation, this clarifies that the increase of execution time is caused by communication, and not by the additional calculations.

In conclusion, communication and calculations can be perfectly overlapped using MPI, additionally confirmed by profiling the application. However, the implementation shows an improvable communication pattern for $K > 4$. With the NCCL-implementation on the contrary, communication and calculations are not fully overlapping. Anyway, the topology-aware communication patterns show clear benefits for the simulation with more than 4 GPUs involved. For $K = 2$, highlighted in figure 3.6, the MPI-implementation is slightly outperforming the NCCL-implementation (factor 1.38 vs. 1.51). With only two GPUs taking part in the simulation, the NCCL's topology-awareness cannot improve communication. In this case overlapping of communication and calculations is much more important.

From a view point of weak scaling, an improved performance is desirable, especially for a large number of involved GPUs. However, regarding the required all-to-all communication the performance metrics are not surprising. Nevertheless, the application enables the simulation of the nonlinear signal propagating of a huge number of spatial modes and a large frequency range, which was not possible so far. Improved performance of the MPI implementation can be expected when decoupling the CPU-GPU control flow. With the future availability of MPI-GDS [159], the asynchronous send operations can be triggered directly after the squared absolute values are computed, leading to even better hiding of the communication. In addition, also the optimization of collective operations is under investigation [17, 18]. Therefore, future library implementations offer the potential to further improve the performance of the proposed implementation.

3.6 Conclusions

In this chapter we have considered the simulation of nonlinear signal propagation in multimode fibers and presented a GPU implementation for the RK4IP algorithm. For this implementation we used the following advanced GPU programming techniques to achieve an efficient implementation.

- By asynchronously copying the data for each mode, the computations can start as soon as the data for the first mode is transferred. This reduces the waiting time before and after the simulation and decreases the walltime.
- By using one stream per mode we achieve a better possibility to hide latencies. Calculations in different streams can be done independently from each other, so they are optimal for hiding latencies.
- For the nonlinear operator we use a so called tiled pattern approach. With this and the use of the fast shared memory, we reduce the effort for this operator.
- We can also store the interaction matrix into the fast constant memory. This reduces the access time for the interaction matrix.

We could show that the algorithm scales poorly on the CPU, while the runtime scales almost linearly with the number of spatial modes when executed on a GPU. A maximum acceleration of 51.0 was achieved for 15 spatial modes, indicating that the GPU significantly outperforms the CPU implementation. So if a GPU is used for acceleration, one can benefit from the advantages of the RK4IP method without having a significant disadvantage due to the number of modes propagating.

Since the problem size quickly becomes too large for a GPU, we have continued to present a multi-GPU approach. Beside the use of NCCL, we also presented an own MPI approach to illustrate how to hide as much communication with computations as possible.

While MPI shows performance benefits for a few used GPUs, the implementation clearly profits from NCCL topology-awareness if more than 4 GPUs are involved in the simulation.

The future availability of MPI-GDS [159], could improve the asynchronous communication in our RK4IP implementation in chapter 3. In addition, also the optimization of collective operations which are under investigation [17, 18] could improve MPI communications in the RK4IP and SPH implementation.

4

Particle-based simulation of flow in porous media

In this chapter the pore-scale resolved flow in porous media is considered with the help of a particle based simulation. The GPU implementation is a challenge because the chosen smoothed particle hydrodynamics (SPH) method leads to unstructured data structures and data accesses and is also a very computationally demanding method. Section 4.1 describes in which application areas this simulation is needed. After the mathematical model is described in the following section 4.2 the numerical method is described in section 4.3. The following sections 4.3.2-4.3.5 address further details of the numerical solution of the problem, such as time integration and boundary conditions. The implementation is based on the general particle simulation toolbox HOOMD-blue, which was extended by the SPH method. How our SPH extension is embedded in HOOMD-blue and which infrastructure can already be used from HOOMD-blue is explained in section 4.4.1. The challenges especially for the GPU implementation as well as different ways to improve the implementation are discussed in sections 4.4.2 and 4.4.3. Finally, these improvements are compared in section 4.5 and the scaling for the CPU and GPU implementation is examined.
The following results have been partially submitted to [128].

4.1 Motivation

Convective flows in porous media have taken the central position in many fundamental heat transfer analyzes and have gained considerable attention in recent decades. This is due to their wide range of applications in high-performance insulation of buildings, chemical-catalytic reactors, grain storage and geophysical problems such as frost heave. Porous media are also of interest in connection with the underground dispersion of pollutants, solar energy collectors and geothermal energy systems.

In recent years, the calculation of effective physical properties of porous media based on tomographic data and pore scale-resolved simulations is often complementing and even replacing physical experiments. Thereby, boundary value problems are formulated directly on the scale of representative unit cells which are obtained from image-based characterization techniques like X-Ray Computed Tomography (XRCT). However, the use of CT scans of microstructures of porous materials such as rocks or soils leads to problems when using grid-based methods such as the finite element method because the complex pore morphologies render meshing and remeshing processes impractically. On the other hand, the use of matrix-free finite differences also leads to problems, since it leads to a serious limitation in the simulation of complex flow processes, e.g., in case of multiphase flow with a significant number of internal interfaces between wetting and non-wetting fluid phases or when investigating larger Reynolds numbers flow. Regarding the mentioned problems, the mesh-free smoothed particle hydrodynamics method is an interesting alternative simulation method. SPH is a Lagrangian particle method, first formulated by Monaghan and Gingold [59] and Lucy [98] for applications in astrophysics. Later it was successfully applied to a wide variety of problems in other fields of physics and mechanics, such as fluid dynamics and solid mechanics. Due to its Lagrange formulation, SPH offers several advantages for the numerical simulation of fluid flows with different Reynolds numbers. Furthermore, the meshless nature of SPH makes the construction of the considered boundary value problems as well as the discretization of CT-generated microstructures comparatively easy compared to mesh-based methods.

However, the main limitation of SPH lies in its high demand for computational resources. Furthermore, using high-resolution CT imaging of real porous materials leads to very large domains. Therefore, in [128] is shown, how a SPH approach is implemented the highly optimized and parallelized tool HOOMD-blue [14, 61], where the author's primary contribution was the GPU implementation and the improvement of the MPI implementation.

In this thesis we limit ourselves to single phase flow applications and begin with describing the mathematical model in the following.

4.2 Mathematical model

The flow of a material such as a gas, single phase liquid or solid is called single phase flow if it is made of exactly one material and not a combination of different materials. Since this chapter focuses on the challenge and improvement of the implementation of the particle-based method and the model and methodology should not be changed, we refer to [16, 21] for the derivation of the physical equations and start directly with the resulting PDE. The single phase flow is described by the conservation of mass and momentum, so the following equations must be solved

$$\dot{\rho} = -\rho\,\nabla\cdot\mathbf{v}, \tag{4.1}$$

$$\rho\,\dot{\mathbf{v}} = \rho\left(\frac{\partial\mathbf{v}}{\partial t} + \nabla\,\mathbf{v}\cdot\mathbf{v}\right) = -\nabla\,p + \nabla\cdot\,(\mu\,\nabla\,\mathbf{v}) + \rho\,\mathbf{b}. \tag{4.2}$$

Equation (4.1) is in fact the mass continuity equation and equation (4.2) is known as the Navier-Stokes equation or momentum continuity equation. Here, $\rho(\mathbf{x},\,t)$ is the density field, $\mathbf{v}$ are the velocities, $p(\mathbf{x},\,t)$ is the pressure field, $\mathbf{b}$ are body force densities and μ is the dynamic viscosity of the fluid.

4.3 Numerical Approximation with SPH

The method chosen here for the numerical discretization is SPH, which is a remarkably versatile and simple approach for numerical fluid dynamics. SPH is a Lagrange method, i.e., the coordinates are moving with the fluid. It can provide a large dynamic range in spatial resolution and density, as well as an automatically adaptive resolution, which are unmatched in Eulerian methods. At the same time, SPH has excellent conservation properties, not only for energy and momentum, but also for angular momentum. The latter is not automatically guaranteed in Eulerian schemes, even though it is usually fulfilled at an acceptable level for well-resolved flows. When coupled to self-gravity, SPH conserves the total energy exactly, which is again not manifestly true in most mesh-based approaches to hydrodynamics. Thanks to its completely mesh-free nature, SPH can easily deal with complicated geometric settings and large regions of space that are completely devoid of particles. This is especially important for our application of CT scans. Implementations of SPH in a numerical code tend to be comparatively simple and transparent. At the same time, the scheme is characterized by remarkable robustness. For example, negative densities or temperatures, sometimes a problem in mesh-based codes, can not occur in SPH by construction [95].
In a nutshell, the SPH method works as follows: The fluid and the solid phases are represented by an ensemble of particles, each of which carries mass, momentum and other properties (such as pressure, internal energy, etc.), see figure 4.1. The temporal evolution is determined by the equation of motion plus additional equations to modify the hydrodynamic properties of the particles. Hydrodynamic observation values are obtained by a local averaging process.

In the following section the principals of the SPH method are discussed and requirements for good kernel functions are described.

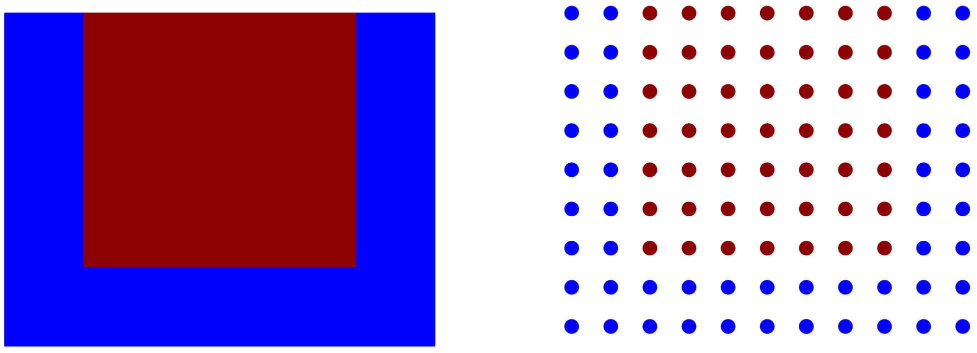


Figure 4.1: Fluid (red) and solid (blue) are represented by particles.

4.3.1 Smoothed particle hydrodynamics

In SPH, the discretization of the governing partial differential equations gives rise to a set of interacting collocation points (particles) $\mathbf{x}_i$ with mass m_i. At each integration point, scalar field functions $f(\mathbf{x})$ are locally interpolated using convolution with the Dirac function δ

$$f(\mathbf{x}) \;=\; \int_\Omega f(\mathbf{x}')\,\delta(\mathbf{x}-\mathbf{x}')\,\mathrm{d}\mathbf{x}', \tag{4.3}$$

which can be approximated by a kernel function W with a compact support κh (figure 4.2).

$$f(\mathbf{x}) \;\approx\; \int_\Omega f(\mathbf{x}')\,W(\mathbf{x}-\mathbf{x}',h)\,\mathrm{d}v = f_h(\mathbf{x}). \tag{4.4}$$

In the context of numerical computations the kernel function W is required to have compact support with h representing a characteristic finite width of the compact support. Following common practice, we refer to W as smoothing kernel and h is referred to as smoothing length. The integral representation f_h is considered to be a reproducing kernel approximation of f if W meets the Dirac delta condition

$$\lim_{h\to 0} W(\mathbf{x},h) = \delta(\mathbf{x}),$$

by virtue of

$$\lim_{h\to 0} f_h(\mathbf{x}) = f(\mathbf{x}).$$

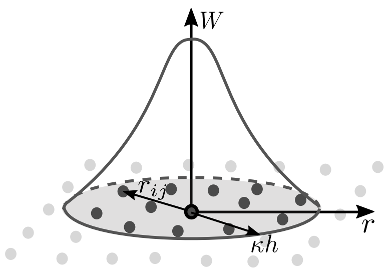


Figure 4.2: Compact support for the kernel function W.

The quadrature rule is used to discretize this integral in SPH methods as a nodal integration equivalent to a middle Riemann sum. In case of equation (4.4) this leads to

$$f_i \;=\; \sum_{j=1}^{N_{neigh}} f_j\,W(\mathbf{x}_i-\mathbf{x}_j,\,h)\,V_j, \tag{4.5}$$

where V_j is the volume of a particle centered at position $\mathbf{x}_j$. The volume V_j can be interpreted as the discrete equivalent of the volume element $\mathrm{d}v$. Using the density $\rho_j =$

$\rho(\mathbf{x}_j)$, the volume of particle j can be computed as $V_j = m_j/\rho_j$, where m_j describes the lumped mass of particle j. This representation is called discrete SPH form. Thereby the kernel representation converts into a spatial discretization and continuous field functions $f(\mathbf{x})$ transform into particle properties $f_k = f_h(\mathbf{x}_k)$. Continuous differential operators are discretized analogously and transform into short-range interaction forces [112, 140].

Kernel functions To ensure that the SPH method is consistent for all application of the method to certain equations, the kernel functions must meet certain requirements. In the following, various properties that a kernel function should fulfill are defined. Afterwards it is shown why these properties are useful and necessary.

- A kernel function must be normalized:

$$\int_\Omega W(\mathbf{x}-\mathbf{x}', h) d\mathbf{x}' = 1 \tag{4.6}$$

- A kernel function should have compact support. In general, the compact support is defined by the smoothing length h and a scaling factor κ that determines the spread of the specified smoothing function. So the compact support means

$$W(\mathbf{x}-\mathbf{x}', h)) = 0 \qquad \text{for } \|\mathbf{x}-\mathbf{x}'\|_2 > \kappa h \tag{4.7}$$

- A kernel function should be non-negative:

$$W(\mathbf{x}-\mathbf{x}', h)) \geq 0 \text{ in the compact support area of } \|\mathbf{x}-\mathbf{x}'\|_2 \leq \kappa h \tag{4.8}$$

- A kernel function should be monotonically decreasing:

$$W(\mathbf{x}, h) > W(\mathbf{x}', h) \text{ for all } \mathbf{x},\, \mathbf{x}' \text{ with } \|\mathbf{x}\|_2 > \|\mathbf{x}'\|_2 \tag{4.9}$$

- A kernel function should satisfy the Dirac delta function condition as $h \to 0$:

$$\lim_{h\to 0} W(\mathbf{x}, h) = \delta(\mathbf{x}) \tag{4.10}$$

- A kernel function should be an even (symmetric) function:

$$W(\mathbf{x}, h) = W(-\mathbf{x}, h) \tag{4.11}$$

- A kernel function should be sufficiently smooth.

In order to quantify the order of completeness of the reproducing kernel approximation, that is the ability of f_h to exactly reproduce a given polynomial f of degree m for finite width h, we introduce the Taylor expansion of $f(\mathbf{x}')$ about point $\mathbf{x}$ given as

$$\Phi(\mathbf{x}') = \Phi(\mathbf{x}) + \frac{\partial \Phi}{\partial \mathbf{x}} \cdot (\mathbf{x}' - \mathbf{x}) + \frac{1}{2}(\mathbf{x}' - \mathbf{x})^\top \frac{\partial^2 \Phi}{\partial \mathbf{x}^2}(x' - \mathbf{x}) + \mathcal{O}(h^3), \tag{4.12}$$

noting that $\mathcal{O}(h^3) = \mathcal{O}(\|\mathbf{x}' - \mathbf{x}\|_2^3)$, where $\| \cdot \|_2$ denotes the Euclidean norm and $f(\mathbf{x}')$ is assumed smooth over the finite interval $(\mathbf{x}', \mathbf{x})$. Substituting the expression in equation (4.12) into equation (4.4) leads to

$$\begin{aligned}\Phi_h(\mathbf{x}') = & \Phi(x) \int_\Omega W(\mathbf{x} - \mathbf{x}', h) \mathrm{d}\mathbf{x}' + \frac{\partial \Phi}{\partial \mathbf{x}} \cdot \int_\Omega (\mathbf{x}' - \mathbf{x}) W(\mathbf{x} - \mathbf{x}', h) \mathrm{d}\mathbf{x}' \\ & + \frac{1}{2} \frac{\partial^2 \Phi}{\partial \mathbf{x}^2} : \int_\Omega (\mathbf{x}' - \mathbf{x})(\mathbf{x}' - \mathbf{x})^\top W(\mathbf{x} - \mathbf{x}', h) \mathrm{d}\mathbf{x}' + \mathcal{O}(h^3).\end{aligned} \tag{4.13}$$

It is evident from equation (4.13) that for f_h to be complete to zeroth order, W has to satisfy the zeroth order completeness condition, hereafter referred to as normalization condition (4.6). The second term on the right hand side of equation (4.13) implies that first order completeness requires the first-order moment condition, hereafter referred to as symmetry condition,

$$\int_\Omega \mathbf{x} W(\mathbf{x}, h) \mathrm{d}\mathbf{x}' = 0$$

to be satisfied. This is also fulfilled due to characteristic (4.11).

Other kernel properties must be physically justified. Characteristic (4.8) ensures the robustness of the method. This characteristic guarantees that, e.g., no negative pressures can occur. The influence of the particles is given by the field functions but not by the kernel weighting. Furthermore characteristic (4.9) causes that less distant particles have a higher influence than particles with a higher distance. Finally, characteristic (4.10) ensures method consistency and characteristic (4.7) must be satisfied for the method to be finite.

Example: There are many commonly used kernel functions. A family of kernel functions often used for SPH applications are the so-called Wendland kernels. As an example, we consider here the Wendland C4 kernel in one dimension, which is defined as

$$\begin{aligned} W(r) &= \frac{3}{4}\left(1 - \frac{r}{2}\right)^5 \left(2r^2 + 2.5r + 1\right) && \text{for } 0 \leq r \leq 2 \\ &= 0 && \text{for } r > 2 \end{aligned}$$

where $r = |x - x_i|$ for fixed x_i. The kernel function as a function of $x - x_i$ is shown in figure 4.3 and its derivative in figure 4.4.

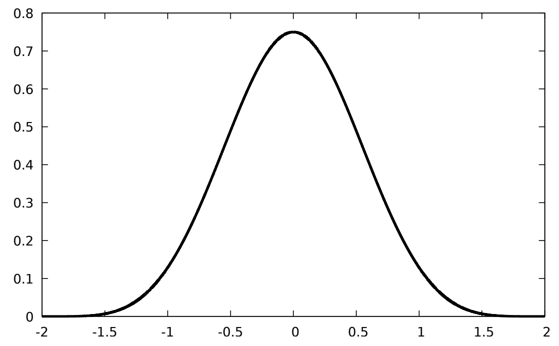


Figure 4.3: Wendland C4 kernel function.

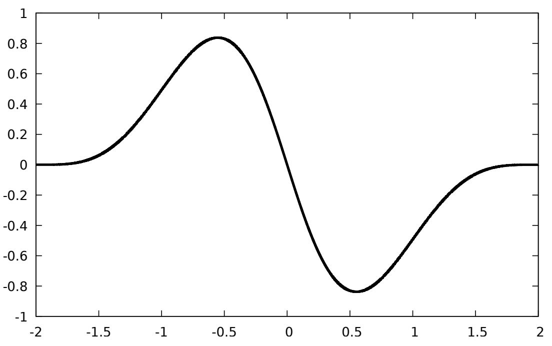


Figure 4.4: First derivative of Wendland C4 kernel function.

Approximation of derivatives We need to approximate spatial differential operators that are inherent in balance equations on the basis of the reproducing kernel approximation. We evaluate the approximation of the gradient of a scalar field f by means of applying equation (4.4) to∇f, which leads to

$$\nabla f(\mathbf{x}) = \int_\Omega \frac{\partial f(\mathbf{x}')}{\partial \mathbf{x}'} W(\mathbf{x}-\mathbf{x}', h)\mathrm{d}\mathbf{x}'.$$

Using integration by parts this can be rewritten to

$$\nabla f(\mathbf{x}) = \int_\Omega \frac{\partial}{\partial \mathbf{x}'}\left[f(\mathbf{x}')W(\mathbf{x}-\mathbf{x}', h)\right]\mathrm{d}\mathbf{x}' - \int_\Omega f(\mathbf{x}')\frac{\partial W(\mathbf{x}-\mathbf{x}', h)}{\partial \mathbf{x}'}\mathrm{d}\mathbf{x}'.$$

Application of Gauss theorem to the first term gives

$$\nabla f(\mathbf{x}) = \int_\Gamma \left[f(\mathbf{x}')W(\mathbf{x}-\mathbf{x}', h)\right]\mathbf{n}(\mathbf{x}')\mathrm{d}\mathbf{a} - \int_\Omega f(\mathbf{x}')\frac{\partial W(\mathbf{x}-\mathbf{x}', h)}{\partial \mathbf{x}'}\mathrm{d}\mathbf{x}'.$$

where $\mathbf{n}$ denotes the unit normal vector to the boundary of the computation domain $\Gamma = \partial\Omega$. Since the kernel W has a compact support, the integral along the boundary in the last equation evaluates to zero if the focal point $\mathbf{x}$ lies at some distance $\|\mathbf{x}_B - \mathbf{x}\| > \kappa h$, $\forall \mathbf{x}_B \in \Gamma$ from the boundary Γ. If the bulk volume of Ω is sufficiently large, the latter condition applies to the majority of focal points. Neglecting the boundary integral and taking the kernel radial symmetry into account, which implies the antisymmetry

$$\frac{\partial W(\mathbf{x}-\mathbf{x}', h)}{\partial \mathbf{x}'} = -\frac{\partial W(\mathbf{x}-\mathbf{x}', h)}{\partial \mathbf{x}}$$

yields the approximate expression for the gradient of a scalar field

$$\nabla f(\mathbf{x}) = \int_\Omega f(\mathbf{x}')\frac{\partial W(\mathbf{x}-\mathbf{x}', h)}{\partial \mathbf{x}}\mathrm{d}\mathbf{x}' = \int_\Omega f(\mathbf{x}')\nabla W(\mathbf{x}-\mathbf{x}', h)\mathrm{d}\mathbf{x}',$$

where the spatial gradient operator is now observed to act on the continuously differentiable kernel function. We use equation (4.5) to transfer this into the discrete SPH form and obtain

$$\nabla f_i = \sum_{j=1}^{N_{neigh}} f_j \frac{m_j}{\rho_j} \nabla W(\mathbf{x}_i - \mathbf{x}_j, h).$$

We introduce the shorthand notations

$$W_{ij} = W(\mathbf{x}_i - \mathbf{x}_j, h) \quad \text{and} \quad \mathbf{r}_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|_2.$$

Due to radial symmetry, the spatial gradient of the kernel may be evaluated as

$$\nabla W(\mathbf{x}_i - \mathbf{x}_j, h) = \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \frac{\mathbf{x}_i - \mathbf{x}_j}{\mathbf{r}_{ij}}, \qquad \text{where} \quad \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} := \left. \frac{\partial W(\mathbf{r}, h)}{\partial \mathbf{r}} \right|_{\mathbf{r}=\mathbf{r}_{ij}}.$$

With this notation we write the gradient as

$$\nabla f_i = \sum_{j=1}^{N_{neigh}} f_j \frac{m_j}{\rho_j} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \frac{\mathbf{x}_i - \mathbf{x}_j}{\mathbf{r}_{ij}}. \tag{4.14}$$

In the same way we get the SPH discretization for the divergence operator as

$$\nabla \cdot f_i = \sum_{j=1}^{N_{neigh}} f_j \frac{m_j}{\rho_j} \cdot \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \frac{\mathbf{x}_i - \mathbf{x}_j}{\mathbf{r}_{ij}}. \tag{4.15}$$

4.3.2 Discretization

In the following we want to derive an SPH discretization for the single phase flow simulation , based on the report by Monaghan [113]. Note that there is not always only one SPH discretization for a given problem. To derive a discretization for our given problem, we repeat the equations to be solved

$$\dot{\rho} = -\rho \nabla \cdot \mathbf{v}, \tag{4.1 revisited}$$

$$\rho \dot{\mathbf{v}} = \rho \left(\frac{\partial \mathbf{v}}{\partial t} + \nabla \mathbf{v} \cdot \mathbf{v} \right) = -\nabla p + \nabla \cdot (\mu \nabla \mathbf{v}) + \rho \mathbf{b}. \tag{4.2 revisited}$$

First we want to derive a discretization for the mass conservation equation (4.1), respectively we want to derive a calculation for the density. For this we show two possibilities. Instead of applying the SPH method to equation (4.1), equation (4.5) is usually simply used to calculate the current pressures. This approach is called summation kernel interpolation, and the density results in

$$\rho_i = \sum_{j=1}^{N_{neigh}} m_j W_{ij}. \tag{4.16}$$

Another approach would be to discretize equation (4.1) using equation (4.15). In this approach, the derivative is found by an exact derivation of an approximate function. However, this form of the derivative does not vanish if the field function $f(\mathbf{x})$ is constant. A simple way to ensure that it does vanish if $f(\mathbf{x})$ is constant is to write

$$\begin{aligned}\frac{\partial f}{\partial \mathbf{x}} &= \frac{1}{\Phi} f \frac{\partial \Phi}{\partial \mathbf{x}} + \frac{\partial f}{\partial \mathbf{x}} - \frac{1}{\Phi} f \frac{\partial \Phi}{\partial \mathbf{x}} \\ &= \frac{1}{\Phi}\left(f\frac{\partial \Phi}{\partial \mathbf{x}} + \Phi \frac{\partial f}{\partial \mathbf{x}} - f\frac{\partial \Phi}{\partial \mathbf{x}} \right) \\ &= \frac{1}{\Phi}\left(\frac{\partial (\Phi f)}{\partial \mathbf{x}} - f\frac{\partial \Phi}{\partial \mathbf{x}} \right)\end{aligned}$$

where Φ is any differentiable function. The SPH form is

$$\frac{\partial f_i}{\partial \mathbf{x}} = \frac{1}{\Phi_i} \sum_{j=1}^{N_{neigh}} m_j \frac{\Phi_j}{\rho_j} (f_j - f_i) \frac{\partial W_{ij}}{\partial \mathbf{x}_i},$$

which vanishes if f is constant. In this expression, different choices of Φ lead to different consistent SPH discretizations of the derivative that can be found in the literature. We chose $\Phi = \rho$, insert $\nabla \cdot v$ and obtain

$$\dot{\rho}_i = \sum_{j=1}^{N_{neigh}} m_j (\mathbf{v}_i - \mathbf{v}_j) \cdot \frac{\mathbf{x}_{ij}}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \, , . \tag{4.17}$$

After we have derived two discretization possibilities for the pressure, we want to discretize (4.2). We start with the first term from equation (4.2) and derive an SPH formulation for

$$\rho \dot{\mathbf{v}} = -\nabla p. \tag{4.18}$$

As described above, the direct exploitation of equation (4.14) does not guarantee that the derivative of constant functions will vanish. Therefore we use the following trick

$$\nabla \phi = \nabla \left(\rho \frac{\phi}{\rho} \right) = \rho \nabla \left(\frac{\phi}{\rho} \right) + \frac{\phi}{\rho} \nabla \rho. \tag{4.19}$$

We divide equation (4.18) by ρ, substitute the gradient by equation (4.19) and apply the SPH formulation to both terms. For the first term this results in

$$\nabla \left(\frac{p_i}{\rho_i} \right) = \sum_{j=1}^{N_{neigh}} m_j \frac{\frac{p_j}{\rho_j}}{\rho_j} \nabla W_{ij} = \sum_{j=1}^{N_{neigh}} m_j \frac{p_j}{\rho_j^2} \nabla W_{ij}$$

and the second term yields

$$\frac{p}{\rho^2}\nabla\rho_i = \frac{p_i}{\rho_i^2}(\sum_{j=1}^{N_{neigh}} m_j \nabla W_{ij}).$$

All in all, we get

$$\begin{aligned}\dot{v}_i =& -\sum_{j=1}^{N_{neigh}} m_j \frac{p_j}{\rho_j^2}\nabla W_{ij} - \frac{p_i}{\rho_i^2}(\sum_{j=1}^{N_{neigh}} m_j \nabla W_{ij}) = -\sum_{j=1}^{N_{neigh}} m_j \left(\frac{p_j}{\rho_j^2}+\frac{p_i}{\rho_i^2}\right)\nabla W_{ij} \\ =& -\sum_{j=1}^{N_{neigh}} m_j \left(\frac{p_j}{\rho_j^2}+\frac{p_i}{\rho_i^2}\right)\frac{x_{ij}}{\mathbf{r}_{ij}}\frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}}. \end{aligned} \tag{4.20}$$

The second term of the momentum continuity equation (4.2) describes the viscous interaction forces between particles and reads as

$$\rho\dot{\mathbf{v}} = \nabla \cdot (\mu \nabla \mathbf{v}). \tag{4.21}$$

Here, second derivatives occur. Therefore we need an SPH discretization for second derivatives. To get a discretization for this we follow the paper by Brookshaw [36] and start with the Taylor series to develop a discretization for the Laplace operator:

$$T(\mathbf{x}') = T(\mathbf{x}) + \nabla T(\mathbf{x})\cdot(\mathbf{x}'-\mathbf{x}) + \frac{1}{2}\frac{\partial^2 T(\mathbf{x})}{\partial x_s \partial x_k}(\mathbf{x}'-\mathbf{x})(\mathbf{x}'-\mathbf{x}) + \mathcal{O}(\mathbf{x}'-\mathbf{x})^3$$

We write the mixed and second derivatives on the left and everything else on the right side and get:

$$\frac{\partial^2 T(\mathbf{x})}{\partial x_s \partial x_k}(\mathbf{x}'-\mathbf{x})(\mathbf{x}'-\mathbf{x}) = -2\left(T(\mathbf{x}) - T(\mathbf{x}') + \nabla T(\mathbf{x})\cdot(\mathbf{x}'-\mathbf{x})\right) + \mathcal{O}(\mathbf{x}'-\mathbf{x})^3 \tag{4.22}$$

Next we neglect terms of third and higher orders, we multiply this with

$$\frac{(\mathbf{x}'-\mathbf{x})\nabla W(\mathbf{x}'-\mathbf{x},h)}{\|\mathbf{x}'-\mathbf{x}\|_2^2},$$

integrate over $\mathbf{x}'$ and note that

$$\begin{aligned}\int_\Omega (\mathbf{x}'-\mathbf{x})\frac{(\mathbf{x}'-\mathbf{x})\nabla W(\mathbf{x}'-\mathbf{x},h)}{\|\mathbf{x}'-\mathbf{x}\|^2}\mathrm{d}\mathbf{x}' =& 0, \\ \int_\Omega (\mathbf{x}'-\mathbf{x})_s(\mathbf{x}'-\mathbf{x})_k\frac{(\mathbf{x}'-\mathbf{x})\nabla(\mathbf{x}'-\mathbf{x},h)}{\|\mathbf{x}'-\mathbf{x}\|^2}\mathrm{d}\mathbf{x}' =& \delta_{sk},\end{aligned}$$

where $(\cdot)_s$ and $(\cdot)_k$ represent the s-th entry, or k-th entry respectively. Note that we choose this kernel to be spherically symmetric and normalized to unity. This results in only the second derivatives remaining on the left side in equation (4.22), i.e., the Laplace operator, and on the right side the gradients of T vanish:

$$\triangle T(\mathbf{x}) = -2 \int \frac{T(\mathbf{x}) - T(\mathbf{x}')}{\|\mathbf{x}' - \mathbf{x}\|^2} (\mathbf{x}' - \mathbf{x}) \nabla W(\mathbf{x}' - \mathbf{x}, h) \mathrm{d}\mathbf{x}'$$

We then end up with a discrete SPH approximation of the Laplace operator in the form

$$\triangle T(\mathbf{x}_i) = -2 \sum_{j=1}^{N_{neigh}} \frac{m_j}{\rho_j} \frac{T(\mathbf{x}_i) - T(\mathbf{x}_j)}{\|\mathbf{x}' - \mathbf{x}\|^2} (\mathbf{x}_j - \mathbf{x}_i) \nabla W(\mathbf{x}_j - \mathbf{x}_i, h). \tag{4.23}$$

Now that we have a discretization for the Laplace operator, we now consider how this can be applied to the thermal conduction problem, where the conductivity may also show a spatial variation. Using the identity

$$\nabla \cdot (\mu \nabla \mathbf{v}) = \frac{1}{2} \left[\triangle(\mu \mathbf{v}) - \mathbf{v} \triangle \mu + \mu \triangle \mathbf{v} \right] \tag{4.24}$$

we can use our result from equation (4.23) to write down a discretized form of equation (4.21) divided by ρ. The three terms in equation (4.24) can be discretized as

$$\frac{1}{2} \triangle(\mu \mathbf{v}) = - \sum_{j=1}^{N_{neigh}} \frac{m_j}{\rho_j} \frac{\mu_j \mathbf{v}_j - \mu_i \mathbf{v}_i}{\|\mathbf{x}' - \mathbf{x}\|^2} (\mathbf{x}_j - \mathbf{x}_i) \nabla W(\mathbf{x}_j - \mathbf{x}_i, h),$$

$$\frac{1}{2} \mathbf{v} \triangle \mu = -\mathbf{v}_i \sum_{j=1}^{N_{neigh}} \frac{m_j}{\rho_j} \frac{\mu_j - \mu_i}{\|\mathbf{x}' - \mathbf{x}\|^2} (\mathbf{x}_j - \mathbf{x}_i) \nabla W(\mathbf{x}_j - \mathbf{x}_i, h),$$

$$\frac{1}{2} \mu \triangle \mathbf{v} = -\mu_i \sum_{j=1}^{N_{neigh}} \frac{m_j}{\rho_j} \frac{\mathbf{v}_j - \mathbf{v}_i}{\|\mathbf{x}' - \mathbf{x}\|^2} (\mathbf{x}_j - \mathbf{x}_i) \nabla W(\mathbf{x}_j - \mathbf{x}_i, h).$$

The summation of these three terms results in

$$\dot{\mathbf{v}} = \frac{1}{\rho} \nabla \cdot (\mu \, \nabla \mathbf{v}) = \sum_{j=1}^{N_{neigh}} \frac{m_j (\mu_i + \mu_j)(\mathbf{v}_i - \mathbf{v}_j)}{\rho_i \, \rho_j} \left(\frac{1}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \right). \tag{4.25}$$

The SPH discretization of the momentum balance equation results in the motion equation for fluid particle i, where the total force on each particle is described as the sum of body forces $\mathbf{F}_i^B = m_i \mathbf{b}$, pressure interaction forces $\mathbf{F}_{ij}^P$ and viscous interaction forces $\mathbf{F}_{ij}^V$ between particles i and j:

$$m_i \, \dot{\mathbf{v}}_i = \sum_{j=1}^{N_{neigh}} \mathbf{F}_{ij}^P + \sum_{j=1}^{N_{neigh}} \mathbf{F}_{ij}^V + \sum_{j=1}^{N_{neigh}} \mathbf{F}_j^B \tag{4.26}$$

Inserting the equation (4.25) and (4.20) finally yields

$$\begin{aligned}\dot{\mathbf{v}}_i = &- \sum_{j=1}^{N_{neigh}} m_j \left(\frac{p_i}{{\rho_i}^2} + \frac{p_j}{{\rho_j}^2} \right) \frac{\mathbf{x}_{ij}}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \\ &+ \sum_{j=1}^{N_{neigh}} \frac{m_j(\mu_i + \mu_j)(\mathbf{v}_i - \mathbf{v}_j)}{\rho_i\, \rho_j} \left(\frac{1}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \right) + \mathbf{b}.\end{aligned} \tag{4.27}$$

The implemented SPH module follows a weakly compressible scheme, where the pressure is calculated from an equation of state [21],

$$p(\rho) = \frac{\rho_0\, c^2}{\gamma} \left[\left(\frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right], \tag{4.28}$$

where c is the speed of sound, $\rho_0 = \rho(\mathbf{x},\, t_0)$ is the reference density and γ a constant defined according to the problem. $\gamma = 7$ is a common value for quasi-incompressible fluids [21].

Alternative discretizations As already mentioned, there are different SPH discretizations. Another SPH discretization, which is especially good for high Reynolds numbers [2], is the following. Thereby, the pressure interaction forces are discretized by

$$-\frac{1}{\rho}\nabla p = -\frac{1}{m_i} \sum_{j=1}^{N_{neigh}} \left(V_i^2 + V_j^2\right) \left(\frac{\rho_j p_i + \rho_i p_j}{\rho_i + \rho_j} \right) \frac{\mathbf{x}_{ij}}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}},$$

and viscous interaction forces are given by

$$\frac{1}{\rho}\nabla \cdot (\mu \nabla \mathbf{v}) = \frac{1}{m_i} \sum_{j=1}^{N_{neigh}} \left(V_i^2 + V_j^2\right) \frac{2\mu_i \mu_j}{\mu_i \ + \mu_j} \left(\frac{\mathbf{v}_{ij}}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \right).$$

All in all the momentum balance equation is discretized by

$$\begin{aligned}\dot{\mathbf{v}}_i = &-\frac{1}{m_i} \sum_{j=1}^{N_{neigh}} \left(V_i^2 + V_j^2\right) \left(\frac{\rho_j p_i + \rho_i p_j}{\rho_i + \rho_j} \right) \frac{\mathbf{x}_{ij}}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \\ &+ \frac{1}{m_i} \sum_{j=1}^{N_{neigh}} \left(V_i^2 + V_j^2\right) \frac{2\mu_i \mu_j}{\mu_i \ + \mu_j} \left(\frac{\mathbf{v}_{ij}}{\mathbf{r}_{ij}} \frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}} \right) + \mathbf{b}.\end{aligned} \tag{4.29}$$

This SPH discretization is also included in our implementation.

4.3.3 Boundary conditions

For the complete description of the problem we also need the definition of boundary conditions. Boundary conditions occur everywhere where fluid particles are in contact with solid particles. The solid phase is considered to be rigid and therefore fluid-solid interfaces satisfy no-penetration and no-slip boundary conditions. Therefore, we use an implementation, which was introduced by [1] whereby the solid domain (wall) is populated with *dummy* particles. For performance reasons we delete dummy particles that are located outside the kernel support domain of fluid particles.

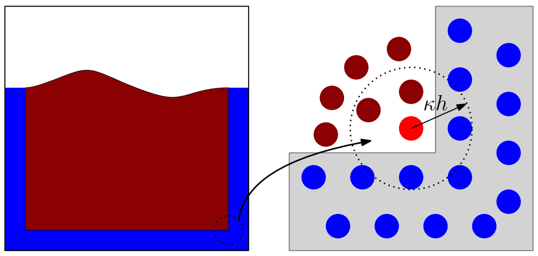


Figure 4.5: Dummy particles (in blue) representing a solid boundary.

The velocity of a dummy particle i is extrapolated from the fluid domain such that

$$\mathbf{v}_i = 2\,\mathbf{v}^{BC} - \frac{\sum_{j=1}^{N_{neigh}} \mathbf{v}_j W_{ij}}{\sum_{j=1}^{N_{neigh}} W_{ij}}\,, \tag{4.30}$$

where $\mathbf{v}^{BC}$ is the wall boundary condition velocity. The pressure of the dummy particle is calculated from the momentum balance, aiming for a no-penetration boundary condition:

$$p_i = \left(\sum_{j=1}^{N_{neigh}} p_j W_{ij} + (\mathbf{b} - \mathbf{a}) \cdot \sum_{j=1}^{N_{neigh}} N_{neigh} \rho_j \mathbf{r}_{ij} W_{ij} \right) \Big/ \left(\sum_{j=1}^{N_{neigh}} W_{ij} \right)\,, \tag{4.31}$$

where $\mathbf{a}$ is the acceleration of the boundary particle and $\mathbf{r}_{ij}$ is the vector between the particles i and j (figure 4.2).

Finally, dummy particle densities are updated by solving equation (4.28) for ρ.

Besides the no-slip and no-penetration boundary conditions, it is also possible to continue domains periodically. This means nothing else than that the left edge is adjacent to the right edge (or upper and lower, front and back) and fluid or ghost particles are set accordingly.

4.3.4 Time integration

Finally, some time step method is needed. Updating particles to the next time step is done with the velocity Verlet time integration method [142], a variation of the Verlet algorithm presented in [160]. We use this method because it is a common choice for particle methods and, compared to methods such as predictor-corrector schemes, has good stability with less computational effort. It is also memory-friendly, since it is not necessary to store previous time steps. In the half-step $(t + \frac{1}{2}\Delta t)$ the velocity is updated as

$$\mathbf{v}(t + \tfrac{1}{2}\,\Delta t) = \mathbf{v}(t) + \tfrac{1}{2}\,\dot{\mathbf{v}}(t)\,\Delta t\,, \tag{4.32}$$

where where $\dot{\mathbf{v}}(t)$ has already been calculated by equation (4.27). In the full step $t + \Delta t$ the position $\mathbf{x}(t)$ and velocity $\mathbf{v}(t)$ are updated as

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t + \tfrac{1}{2}\,\Delta t)\,\Delta t\,, \tag{4.33}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t + \tfrac{1}{2}\,\Delta t) + \tfrac{1}{2}\,\dot{\mathbf{v}}(t + \Delta t)\,\Delta t\,. \tag{4.34}$$

4.3.5 Artificial viscosity

Since the SPH method does not guarantee that nonphysical oscillations occur during the simulation, we add a stabilization method. For this purpose we use the stabilization method presented in [112] and implement an artificial viscosity approach. Here, a dissipative term, the so-called artificial viscosity Π_{ij}, is included in the impulse balance equation for damping non-physical oscillations. The definition of artificial viscosity presented in [112] is the most commonly used,

$$\Pi_{ij} = \begin{cases} \dfrac{-\,\alpha_\Pi\, c_{ij}\,\phi_{ij} + \beta_\Pi\,\phi^2}{\rho_{ij}}, & \mathbf{v}_{ij}\cdot\mathbf{x}_{ij} < 0, \\ 0, & \mathbf{v}_{ij}\cdot\mathbf{x}_{ij} \geq 0, \end{cases} \tag{4.35}$$

with $h_{ij} = \frac{1}{2}(h_i + h_j)$, $c_{ij} = \frac{1}{2}(c_i + c_j)$, $\rho_{ij} = \frac{1}{2}(\rho_i + \rho_j)$,

$$\phi_{ij} = \frac{h_{ij}\mathbf{v}_{ij}\cdot\mathbf{x}_{ij}}{|\mathbf{x}_{ij}|^2 + \varepsilon^2}\,,$$

and α_Π and β_Π are constants. Monaghan [112] recommends the constant values $\alpha_\Pi \approx 1$ and $\beta_\Pi \approx 2$. However the values need to be fitted according to the application. In the presented low Re applications the values $\alpha_\pi = 0.2$ and $\beta_\pi = 0$ were used. The variable $\varepsilon = 0.1\,h$ is introduced to avoid divergence when $|\mathbf{x}_{ij}| \to 0$. Thus the part of the pairwise pressure interaction forces $\mathbf{F}^p_{ij}$ in equation (4.27) can be updated to

$$\mathbf{F}^p_{ij} = -\sum_{j=1}^{N_{neigh}} m_j \left(\frac{p_i}{{\rho_i}^2} + \frac{p_j}{{\rho_j}^2} + \Pi_{ij}\right)\frac{\mathbf{x}_{ij}}{\mathbf{r}_{ij}}\frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}}\,, \tag{4.36}$$

or with the other discretization in equation (4.29) to

$$\mathbf{F}^p_{ij} = -\frac{1}{m_i}\sum_{j=1}^{N_{neigh}} \left(V_i^2 + V_j^2\right)\left(\frac{\rho_j p_i + \rho_i p_j}{\rho_i + \rho_j} + \Pi_{ij}\right)\frac{\mathbf{x}_{ij}}{\mathbf{r}_{ij}}\frac{\partial W_{ij}}{\partial \mathbf{r}_{ij}}\,. \tag{4.37}$$

4.3.6 Schematic SPH single phase flow algorithm

After all required discretizations have been described in the previous sections, they have to be assembled into an algorithm (see algorithm 4.1). Here we present a possible parallel algorithm, i.e., we also show where values must be exchanged between different processes. In order to simulate the discrete SPH model, all neighborhood relations must be calculated first, so the first step of each iteration is to call a nearest neighbor search (NNS) algorithm using the compact support of all operators. The density is required for all particles before other averaged values can be calculated at each time step. Either equation (4.16) or (4.17) can be used. Then the pressures are calculated. Since these can be calculated independently of properties of other particles (see equation (4.28)) this can be done without prior communication. Afterwards the pressures and densities have to be communicated so that they are updated everywhere. Then the boundary conditions are taken into account, i.e. the equations (4.30) and (4.31) are calculated. Alternatively, the calculation of the pressures for the solid particles can be done in `compute_pressure()`. Finally, the pressure interaction forces (see equation (4.25)), the viscous interaction forces together with the stabilization by artificial viscosity (see equation (4.36)) and the body forces are computed. Finally, the next time step for the positions and velocities is calculated with the velocity Verlet time integration (see section 4.3.4).

Algorithm 4.1 (SPH fluid solver).

```
1:  for $t = 0;\ t < T;\ t = t + h$ do
2:      NNS()                                   ▷ see sec. 4.4 or 4.4.3
3:      communicate_neigh_list()
4:      compute_density()                       ▷ eq. (4.16) or (4.17)
5:      compute_pressure()                      ▷ eq. (4.28)
6:      communicate_densiy_prssure()
7:      compute_boundary_conditions()           ▷ eq. (4.30) and eq. (4.31)
8:      compute_forces()                        ▷ eq. (4.36) or eq. (4.37)
9:      update_time_step()                      ▷ eq. (4.32)–(4.34)
10: end for
```

4.4 Implementation aspects and challenges

For our implementation we use the particle simulation toolbox HOOMD-blue [61] as basis and extend it for the SPH method. How our extension is embedded in HOOMD-blue and where we made changes is described in the next section. The author of this thesis has done the implementation and improvement of the GPU aspects as well as novel neighborhood search data structures and algorithms. Afterwards, the general challenges of an SPH implementation for the GPU are described and possible improvements for such an implementation are discussed.

4.4.1 SPH implementation in HOOMD-blue

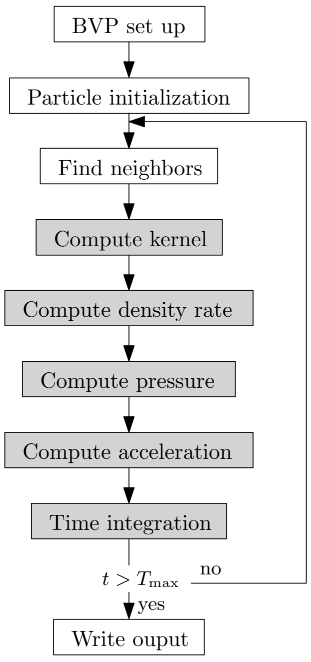


Figure 4.6: SPH workflow. Gray background indicates our own implementation within HOOMD-blue.

HOOMD-blue is designed modularly, for easier modifications and expansions. The boundary value problem (BVP) setup and particle initialization (figure 4.6) are implemented as Python scripts, and the remaining code is C++. Our modifications and main modifications of HOOMD-blue are marked in gray in figure 4.6. Our main changes include the compute and update classes and the particle data structure. These are implemented in C++ and CUDA and we explain them in more detail in the following:

1. *Particle Data*: HOOMD-Blue defines a Particle Data structure intended for Molecular Dynamics applications. We have adapted this for SPH applications. Table 4.1 shows the implemented fields. HOOMD initializes every field of particle data, even if is not used. For memory optimization, only essential fields for single phase flow were active in this study (indicated in table 4.1 with *).

Field	Type	Entries
pos*	Scalar4	$[x_1, x_2, x_3, \text{type}]$
vel*	Scalar4	$[v_1, v_2, v_3, \text{mass}]$
dpe*	Scalar3	$[\rho, p, e]$
kappah*	Scalar	κh
aux1	Scalar3	
aux2	Scalar3	
aux3	Scalar3	
aux4	Scalar3	
accel*	Scalar3	$[a_1, a_2, a_3]$
dpedt*	Scalar3	$[d\rho/dt, dp/dt, de/dt]$
image*	int3	
body*	unsigned int	Body id
tag*	unsigned int	Global tag

Table 4.1: Implemented fields in the Particle Data structure. Essential fields for a single phase flow simulation are marked with *.

2. *Compute*: This module reads the Particle Data fields, computes the neighbor list and calculates density rates (continuity approach) and acceleration. Additionally, the pressure is computed with the defined equation of state. The nearest neighbor search is used directly from the HOOMD-blue package. Furthermore, we have implemented another NNS algorithm for the GPU, which is introduced in section 4.4.3.

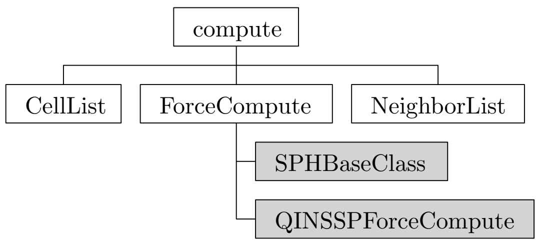


Figure 4.7: Excerpt of the Compute class. Gray boxes mark our own implementations. The module QINSSP presents the implementation of the equations for a quasi incompressible Navier-Stokes single phase flow.

3. *Update*: This class updates Particle Data fields according to the time integration. We implemented the Velocity Verlet integration method as described in section 4.3.4.

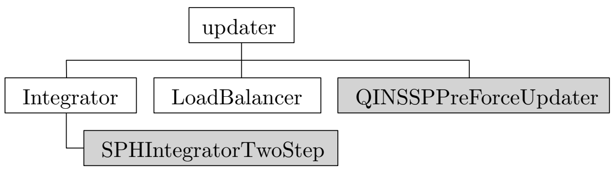


Figure 4.8: Excerpt from updater class. Gray boxes indicates own implementation.

Neighbor list and search in HOOMD-blue The first step in each iteration of an SPH algorithm is the nearest neighbor search (compare algorithm 4.1). Here, all particles that lie within a certain radius around a particle must be found as neighbors. In practice, only a limited maximum number of neighbors is used for the following calculations. HOOMD-blue offers three different NNS algorithms, two based on binned linked-cell list and one based on a linear bounded volume hierarchical (LBVH) tree structure. All three algorithms specify the neighborhood relationships in a so-called Verlet list. This consists of three arrays `n_neigh`, `head_list` and `nlist`. The number of neighbors per particle is given in the array `n_neigh`. The neighbors of every particle are listed in the `nlist` array, first all neighbors of particle 0, then of particle 1 and so on. The position at which the neighboring particles start in the `nlist` array for each particle is listed in the `head_list` array. An example is shown in figure 4.10. This list is based on the same structure as the CSR matrix (which we know from section 2.2). Here, each matrix row corresponds to one particle, and has a 1 for each neighbor particle in the corresponding column. The `col` array from the CSR format corresponds to the `nlist` array from the Verlet list, and the `row` array to the `head_list`. This method is generally well suited for methods such as the SPH method, because it makes it easy to implement loops over all neighbors. Based on the maximum movement of the particles, these algorithms can check whether it is necessary to recreate the list or whether it is still correct. For this purpose, the NNS algorithms search in a radius larger than the corresponding radius of the SPH method. This is useful because the neighborhood search is very complex. The three algorithms in HOOMD-blue are called cell (or cell list), stencil (or stencil cell list) and tree (or LBVH tree) algorithm. The first two are based on a cell list sorting where all particles are segmented into cells. In the following neighbor search, only all particles in neighboring cells have to be considered. The tree algorithm is based on binary tree structures that partition the system based on objects rather than space. This means that the memory required scales with the number of particles in the system rather than the system volume, which can be particularly advantageous for large, sparse systems. For detailed information on the cell list algorithm and its implementation in HOOMD-blue, we referred to [56, 60, 78]. More detailed information on the stencil cell list can be found in a paper by Howard et. al [74] and on the LBVH tree algorithm in [75]. We explore which of these algorithms is the best choice for our use cases in section 4.5.3.

Parallelization and communication in HOOMD-blue HOOMD-blue implements a spatial domain decomposition for the parallelization with MPI. The ghost particles must be communicated between processors, and particles must be migrated between processors when they move across subdomains (figure 4.9 and line 5 in algorithm 4.1). The involved communication is prone to becoming the bottleneck of a simulation. Thus, the domain is decomposed minimizing the inter-domain interfaces. Moreover, each communication is restricted only to the required fields.

HOOMD offers the possibility to transfer the data via P2P (see section 2.3), or if this possibility is not available, to copy the data first to the CPU memory and then to execute the communication.

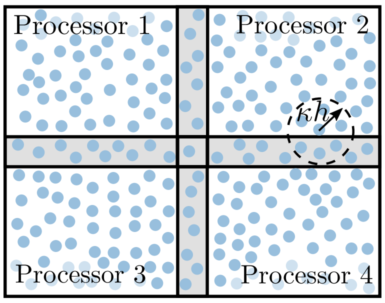


Figure 4.9: MPI spatial domain decomposition. κh denotes the compact support of the kernel function.

4.4.2 Challenges of the implementation

In this section we discuss the challenges of using the standard approach with neighbors in a Verlet list. In section 4.4.3, we alternatively show another approach to work around some of the GPU implementation issues presented here. SPH implementations essentially consist of loops over all particles and for each particle additional nested loops over all neighboring particles. Therefore, the parallelization of each particle loop is quite generic on CPUs, as long as the average number of neighbors is similar for each MPI process. Due to the fine-grained parallelism of GPUs, efficiency is harder to accomplish. Our approach is to run a GPU kernel for each particle loop on the CPU according to the Verlet list. Fluid and solid particles are treated subsequently by separate, sequentially issued kernels, and one GPU thread is used for each particle, which is a standard ad-hoc choice for this Verlet list approach. In general, all GPUs in the domain decomposition should have more or less the same amount of solid and fluid particles to achieve the best efficiency. This is in contrast to several practically relevant scenarios, where large differences between the number of fluid and solid particles are locally possible (see section 4.5). HOOMD-blue implements a heuristic load balancing where in a certain period of steps, the boundaries of the processor domains are adjusted to distribute the particle load close to evenly between them. We choose the default configuration, which has a period of 1000 steps, allows x-, y- and z-direction adjustments and tolerates an imbalance of 1.02. Furthermore, the type (i.e., fluid or solid) of the neighbor is not always the same. For instance, if particle i and j are in the same warp and both have five neighbors, but the first neighbor of i is a solid particle and the first neighbor of j is a fluid particle, warp divergence occur. This could lead to warp divergence (see section 2.2.2), because there are several computations that have only to be done for fluid neighbors. Even if all processes have a similar number of particles, this does not automatically mean that the number of fluid and solid particles is similar. Another situation where warp divergence occurs is when a neighbor particle is not in the support of the kernel function. We recall that the NNS algorithms search in a larger radius to avoid rebuilding the neighbor list at each time step. A good compromise must be found between a buffer for the NNS algorithm, so that it does not have to be executed in every time step, and the resulting warp divergence in the SPH method because several particles are not in the cut-off radius.

Table 4.2 shows these warp divergences for the main GPU kernel functions, with the rows describing different code lines where the warps diverge (for the PSA-HIGH benchmark scenario described in section 4.5). The percentage of warp divergence is measured using the NVIDIA Visual Profiler, where we consider the first call of each kernel in the time loop. The columns correspond to the three main GPU kernel functions of the SPH method and the four rows correspond to different lines in these functions where warp divergence occurs. Here 'N/A' indicates that no further warp divergence occurs, i.e. that the force kernel only has one warp divergence. We see that warp divergence occurs comparatively often, but this cannot be avoided at least not by using the Verlet list. Another possibility is to use a pairs list as we use in section 4.4.3.

Kernel	force eq. (4.37)	dens.+pre. eqs. (4.16)+(4.28)	noslip eqs. (4.30)+(4.31)
Divergent Execution in %	72.5	24.0	38.7
	N/A	11.9	27.3
	N/A	11.9	21.8
	N/A	11.9	10.5
	N/A	2.3	N/A

Table 4.2: Warp divergence, single GPU K40c.

Despite being meshless, the SPH method can be compared to numerical linear algebra operations as described by the analogy to the CSR matrix above, and thus exhibits the same challenges as unstructured matrix codes [86]. For many computations data from all neighbors are necessary, e.g., in the momentum equation (4.2). The nearest neighbor search is a particularly expensive algorithm, which scales with $\mathcal{O}(kN)$, with k being proportional to the average number of neighbors and N being the number of particles [74]. Furthermore, each particle generally has a different number of neighbors, which means that the loop over all neighbors has different numbers of iterations. Hence, if the particles of two threads in a CUDA warp have a different number of neighbors, the thread with fewer neighbors is idle for several iterations (the mismatch between the number of neighbors) of this loop, due to the SIMT execution. We already mentioned above that this is not that critical if all threads per warp have about the same number of neighbors. However, an iteration of this loop usually consists of many instructions, so warp divergence here is quite expensive (see section 2.2.2).

Additionally to this imbalance in the work, the memory accesses are very unstructured. In order to achieve optimal memory access patterns, it is necessary to store not only all particles but also all their neighbors close to each other in memory to achieve optimal data reuse through a single cache line on CPUs and GPUs. Of course, this is generally impossible to achieve, and cache misses sometimes have larger and sometimes smaller adverse effects. For GPUs, the problem is more pronounced because the CUDA warp size (SIMT) far exceeds the SIMD width on CPUs. In terms of latency hiding, it is obviously not possible to use the full capacity of the device if there are fewer particles than CUDA cores.

4.4.3 Improvements for the GPU Implementation

In order to achieve further accelerations, or higher GPU against CPU accelerations, we improve the (GPU-) implementation. We first present two modifications that do not change the neighborhood structure and are therefore relatively easy to implement, which we refer to as Modification 1 and Modification 2. After that we introduce another modification that involves major structural changes to the code because it uses a different neighbor list. The modifications are independent of each other and can each be implemented and used without the others. However, we define the modifications here as successive modifications, so that later modifications include the previous ones or their concepts.

Modification 1 Since density, pressure and energy are the essential properties of a particle, it is common to store them as triples. This results in a so-called Array of Structures which is less suitable for GPU implementations than a Structure of Arrays (see section 2.2.5). The first modification is therefore the use of a Structure of Arrays as it is more efficient to use three arrays instead of one, because one cache line loads more needed data, and less cache lines need to load.

Modification 2 Since there is always a small latency when the CPU launches a GPU kernel, another common strategy for improving the GPU code is to merge as many CUDA kernels as possible. This can result in less overhead on kernel startup, better data reuse, and fewer cache misses. In our implementation, it is possible to combine all kernels that are between the communication phases, which means that the calculation of pressures and densities can be done in the same kernel. Besides eliminating the launch overhead, this leads to the fact that ρ_i, which is otherwise recalculated in the density calculation, can be reused in the pressure calculation, thus eliminating latency since this value is still in the cache and need not be stored to DRAM and loaded again, as in an implementation with multiple individual kernels. Similarly, the calculation of the boundary conditions and the force calculation can be merged. Here the pressure and velocity values calculated in the boundary condition calculation can be reused.

Modification 2 is conceptually independent, but implemented as an incremental modification, so it contains the first modification.

Alternative nearest neighbor search approach

One challenge we described in section 4.4.2 is that the loops over neighbor particles cause warp divergence. To overcome this issue, we implemented another NNS algorithm that uses a pair neighbor list instead of a Verlet neighbor list.

The warp divergence, caused by a loop over all neighbors (see section 4.4.2), can be eliminated by letting a thread represent a pair of neighbors instead of a particle. However, the neighborhood relationships would have to be in a pairs list instead of a Verlet list (see figure 4.10). Thus, we need a function that, according to the NNS, rewrites the

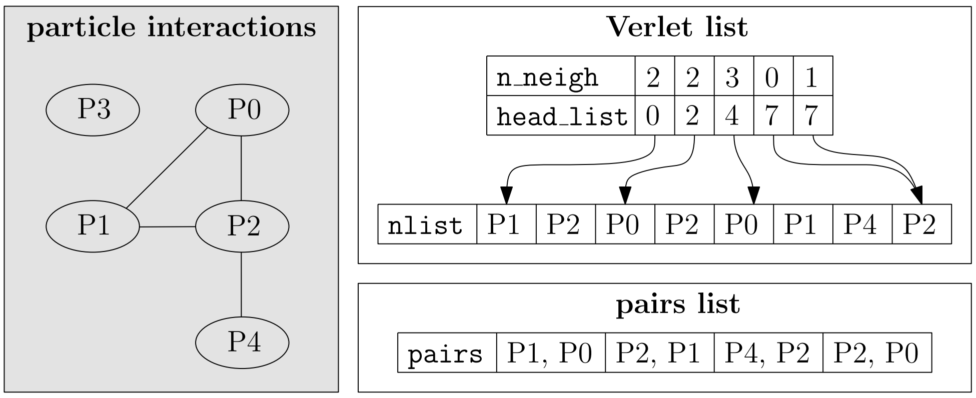


Figure 4.10: Example for a Verlet neighbor list and a pairs neighbor list. The array `n_neigh` contains the number of neighbors per particle, the neighbors of every particle are listed in the `nlist` array and the position at which the neighboring particles start in the `nlist` array for each particle is listed in the `head_list` array.

neighborhood relationships into a pairs list, or an algorithm that writes the neighborhood relationships directly into a pairs list. Since the former leads to an overhead, it is better to implement a new NNS. Furthermore, atomic operations are no longer inefficient on current GPU hardware as they are on older hardware, so NNS algorithms as described in [73] are often faster on current hardware [73]. For this reason, we implement a new NNS algorithm based on the idea of [73] that writes the neighborhood relationships directly to a pairs list. The goal is not only to achieve an acceleration of the NNS algorithm, but also an acceleration of the entire force computations by using the pair structure.

Basic idea of the nearest neighbor search The basic idea is to divide the particles into cells, whereby the complexity of the neighbor search can be reduced to $\mathcal{O}(kN)$, where k is proportional to the average number of neighbors and N is the number of particles. This approach is described, e.g., in [7]. The basic segmentation into cells is identical to the two cell list based HOOMD-blue NNS algorithms. This also means that the HOOMD-blue and our NNS algorithm have the same numerical scalability.

The first step of our nearest neighbor search is to create axis-aligned cells of the size `cell_size_x`, `cell_size_y` and `cell_size_z` so that the whole domain is covered. The cells are numbered continuously from top back left to bottom front right, or x-minor to z-major (see figure 4.11). Furthermore, the particles are sorted into the cells and

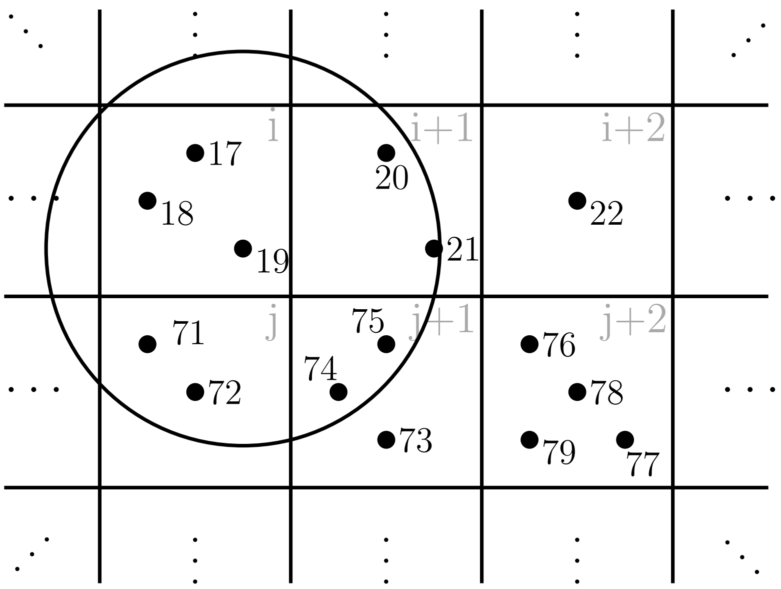


Figure 4.11: The domain is covered by cells numbered from the top back left to bottom right front. The particles are renumbered so that the particles lie linearly in the cells.

renumbered so that the particles in cell 0 have the numbers 0 to *#particles_in_cell_0*−1, in cell 1 the numbers *#particles_in_cell_0* to *#particles_in_cell_0*+*#particles_in_cell_1*−1 (see figure 4.11). Afterwards, the nearest neighbor search is carried out using a local brute force algorithm, which is explained in the next paragraph.

The overall process is the following:

1. Create cells
2. Fill particles into cells / renumber particles by cells, x minor, z major
3. 'Local brute force' search over neighboring cells

Local nearest neighbor search Each warp loads a block of 32 particles from the sorted particles and stores them in shared memory. Then an iteration is performed over all neighboring cells (with potential interaction pairs). That means each warp loads particles from a neighboring cell and compares them with the particles in shared memory. Afterwards a warp loads another 32 particles from a neighboring cell until all particles from neighboring cells are compared with the particles in shared memory. This means that a local brute force NNS is performed for the particles loaded in shared memory. Since all neighbor pairs must and may only exist once, we only search for neighbors that are located in cell numbers that have a smaller cell number than the particles in the shared memory. That means, especially if the pair (p, q) is in the pairs array, the pair (q, p) is not in it. We recall that the cell list algorithms implemented in HOOMD-blue need to search all neighboring cells.

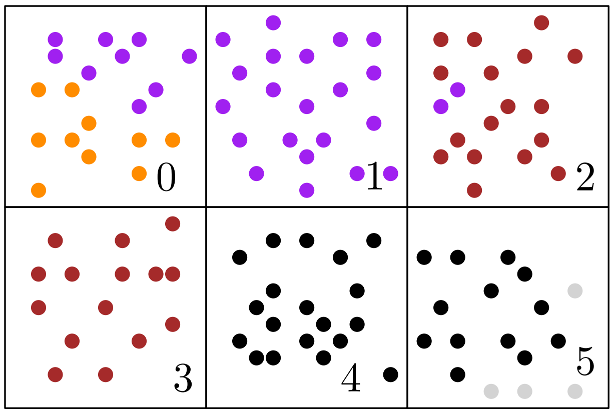


Figure 4.12: Local ‘brute force search’. Black particles are loaded into shared memory, brown are compared in the first iteration, purple in the second and orange in the third iteration.

This is illustrated in figure 4.12. The black particles are those loaded into shared memory by a warp. In the first iteration, the 32 threads of this warp load the brown particles and check if they are adjacent to the black particles. In the second iteration the purple particles are compared and finally the orange particles.

We use the fact that particles, that are located in cells that are adjacent along the x-axis, are linearly stored in the memory (x-minor). Furthermore, because of this sorting we know that all particles loaded into the cache are in cells ’between’ the first and last loaded particle. For the 32 loaded particles this means that for the cell number c_i of particles i ($c_i = c\_idx(x_i, y_i, z_i)$) it holds:

$$c\_idx(x_0, y_0, z_0) \leq c\_idx(x_i, y_i, z_i) \leq c\_idx(x_{31}, y_{31}, z_{31}) \qquad \text{for } i = 0, ..., 31$$

The set of cells that have potentially neighbor particles for the particles in the cache, is the union of all sets of cells with potential neighbors for each particle in the cache. For this we look at figure 4.12 again. The black particles are the 32 particles loaded into the cache. Due to the existing continuous renumbering of the particles by cell, we know that all 32 particles are in cells between the first and last particle (here between cell 4 and 5). Since we are only looking for particle pairs of the type (i, j) with $i < j$, only particles in cells 0, 1, 2 and 3 and the 32 particles loaded particles are considered as neighbors.

Disadvantages of this approach When creating the neighbor list, all threads must know at which position in the pairs array the particle pairs should be written. For this purpose, the maximum number of pairs a warp can find is calculated in advance and a buffer with this size is allocated. However, in order to write the particles continuously into the pairs array all threads within a warp must know which of the other threads in the same warp have found a neighbor pair. We discuss the situation which is illustrated in figure 4.13, where thread 1, 7 and 9 find a neighbor pair and all other threads find none. Each of these threads has to know, at which position it has to write the found pair,

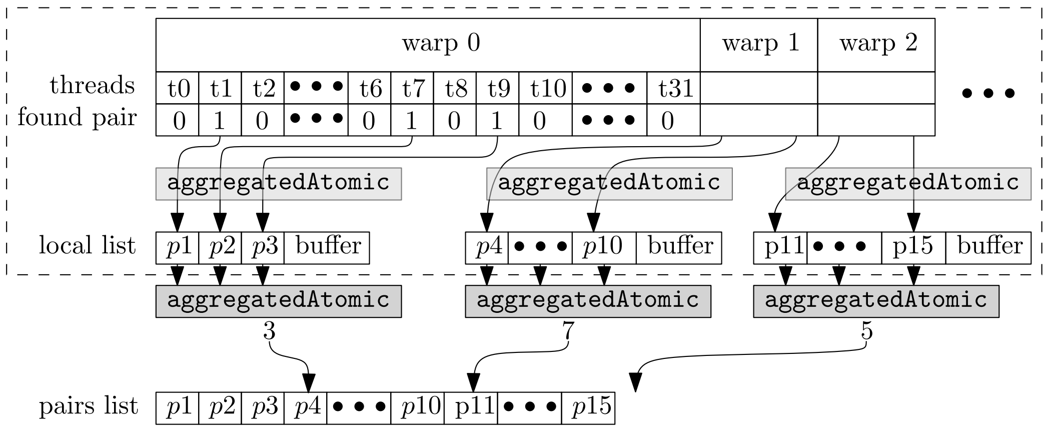


Figure 4.13: Creating pairs list using aggregated atomics.

which can be computed by an atomic add. After that, thread 1 writes its pair to the first position, thread 7 its to the second and thread 9 its to the third position. Additionally the number of pairs found by a warp must be counted to fill the pairs array continuously without any 'gaps' caused by the buffer. To get this information per warp an atomic add could be used. As described in section 2.2.3 this can lead to a performance decrease. But in this case the operation can also be done by aggregated atomic add without losing much performance (see section 2.2.3 and [3, 93]). Such a warp aggregated operation that increases a value by one is shown in listing 4.1. All threads that have found a neighbor pair call the `aggregatedAtomic` function. In line 2 a mask is created, which indicates which threads in this warp have executed the function. Then in the next line an active thread is selected, which then executes an atomic operation to increase the input value by the number of active threads. In line 8 an individual result is calculated for each active thread, so if thread 1, 7 and 9 execute the `aggregatedAtomic` function and the input value is 0, thread 1 gets the value 1 back, thread 7 gets the value 2 and thread 9 gets the value 3. In the best case this reduces the costs of 32 competing `atomicAdd`s to one non-competing `atomicAdd`.

```
1  __device__ int aggregatedAtomic(int *ptr) {
2    int mask = __match_any_sync(__activemask(), (unsigned long long)ptr);
3    int leader = __ffs(mask) - 1;//select a leader who does the real atomic
4    int res;
5    if(lane_id() == leader)
6      res = atomicAdd(ptr, __popc(mask));//leader does the update
7    res = __shfl_sync(mask, res, leader);//get leader's old value
8    res += __popc(mask & ((1 << lane_id()) - 1));//input value increased by 1
9    return res;
10 }
```

Listing 4.1: Aggregated atomic to increase a value by 1. Functions starting with '__' are CUDA intrinsics, see [120].

Further modifications due to the pairs list Because of the new pair structure, the loop over all particles and the loop over all neighboring particles no longer make sense. Instead, all neighbor pairs are iterated over, or from the point of view of the GPU implementation, one thread represents one particle pair. Through this structure it is possible to eliminate even more warp divergence. Sorting all pairs so that all fluid-fluid-particle pairs are listed first, then all fluid-solid pairs, and then all solid-solid pairs can eliminate any divergence that occurs when a specific calculation occurs only for a particular pair combination. Thus, warp divergence only occurs if a pair is not within the cutoff radius of the SPH method.

However, there is another problem with this approach since each thread stands for a pair of particles, many operations must be protected by atomic operations. However, this should not usually result in much performance loss, especially since the first particle of a pair in a warp is different (before sorting by interaction type). Furthermore, the fact that there are many more neighbor pairs than particles means that the CUDA grid is made up of many more threads, offering more ways to hide latency.

4.5 Numerical results

In this chapter we evaluate the new SPH module and the improvements presented. For this purpose we first evaluate which improvements show an efficient implementation on which platform. Then we evaluate the most efficient implementation with respect to weak and strong scalability on both CPU-based and GPU-based HPC clusters. In the end we also evaluate the acceleration potential by using GPUs instead of CPUs.

4.5.1 Benchmark definition

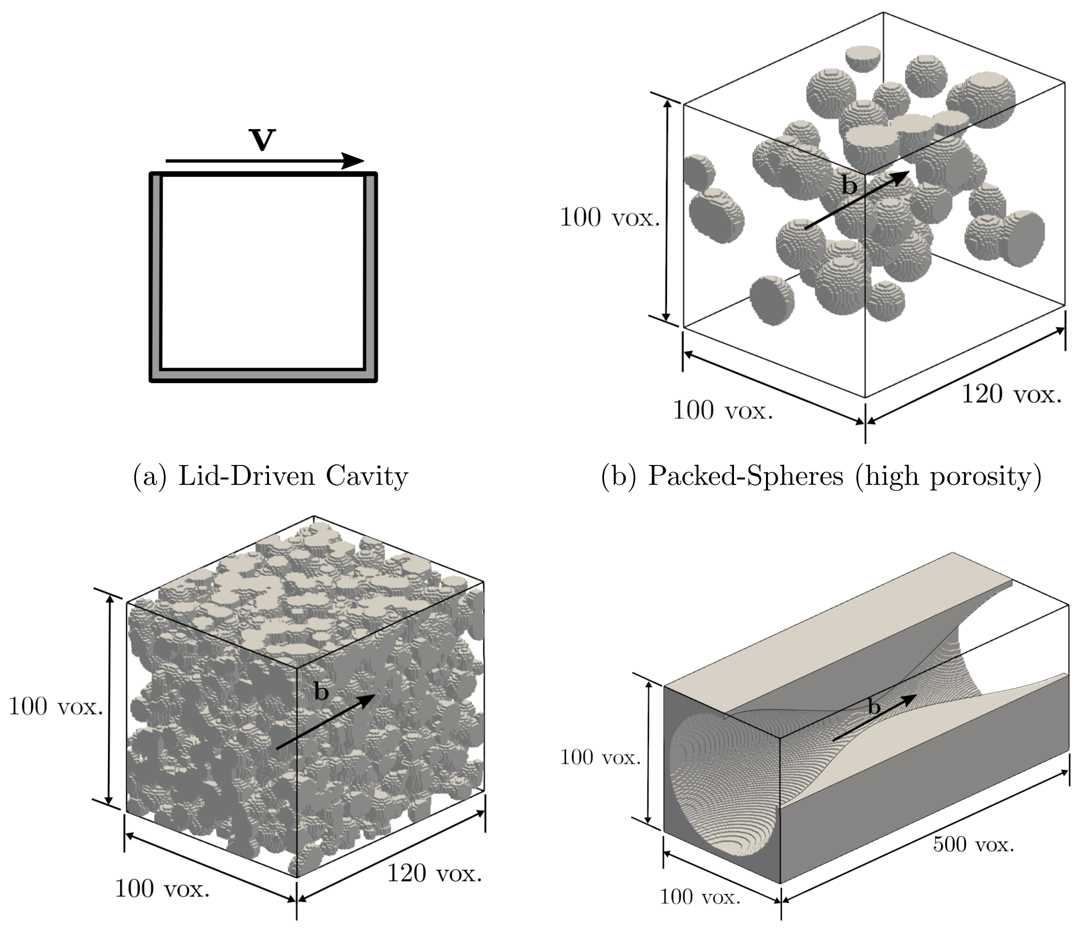
We employ the following four benchmark scenarios, see also figure 4.14:

1. Lid-Driven Cavity (LDC)
2. High-Porosity Packed-Spheres Array (PSA-HIGH)
3. Low-Porosity Packed-Spheres Array (PSA-LOW)
4. Variable Cross-Section Channel (VC)

These benchmark scenarios are common representative choices in fluid dynamics and flow through porous media, and constitute, from best to worst, the anticipated strong scaling behavior of the spatial domain decomposition as explained below. Due to the high particle numbers in the LDC benchmark, the radius of the NNS algorithm is set equal to the cut-off radius of the SPH method. However, this leads to the fact that the NNS has to be performed in every time step. For the other simulations we set the radius of the NNS method 5% larger than the cut-off radius. To compare the NNS method based on the Verlet list and the one based on the pairs list we use the LDC and the PSA-HIGH benchmark. Weak scalability is exemplarily assessed for the PSA-HIGH and PSA-LOW problems, as these scenarios are most representative for the Digital Rock Physics applications we are ultimately interested in. Table 4.3 lists the number of particles for each benchmark problem when assessing strong scalability, and at the same time the baseline for our weak scalability experiments. The number of solid particles is measured after eliminating the solid particles that have no neighbor fluid particles within their compact support κh, i.e., after eliminating solid particles that never influence fluid particles.

	fluid particles	solid particles	total
a) LDC	10.000.000	120.360	10.120.360
b) PSA-HIGH	4.970.912	527.992	5.498.904
c) PSA-LOW	1.071.392	3.061.830	4.133.222
d) VC	2.056.248	389.604	2.442.252

Table 4.3: Number of particles in the benchmark scenarios.



(a) Lid-Driven Cavity

(b) Packed-Spheres (high porosity)

(c) Packed-Spheres (low porosity)

(d) Variable Cross-Section Channel

Figure 4.14: Illustration of the four benchmark scenarios.

HOOMD-blue, similar to many particle codes, measures performance in *time steps per second (TPS)*, with a higher value indicating a better performance. For each benchmark scenario, we report the TPS metric for 100 time steps, and perform file I/O only at the very beginning (to load the geometry etc.) and the end (to save the simulation result).

4.5.2 Hardware details

All CPU experiments are conducted on BinAC, a Tier-3 machine operated by the University of Tübingen as part of the BW-HPC strategy. This cluster consists of 300 nodes. A dual-socket node contains two Intel Haswell E5-2680v4 processors (base frequency 2.4 GHz, 12 cores), leading to a total number of 24 cores per node since we do not use hyperthreading due to diminishing returns. Each node includes 128 GB DDR4 RAM. In addition, this cluster has another 60 nodes with the same setup as the other nodes and also contain two NVIDIA Tesla K80, that we use for the GPU experiments. The K80 design comprises two identical Kepler GPUs per accelerator board, each with its dedicated GDDR memory, for a total of two times two GPUs per node. In summary, the BinAC cluster constitutes the state of the art in GPU clusters as of 2015. Since the Kepler architecture has been superseded by the Pascal and Volta architectures, we also use the

Cray CS-Storm at HRLS. A dual-socket node of this Cray CS-Storm contains two Intel XeonGold 6240 processors (base frequency 2.6 GHz, 18 cores) and eight NVIDIA Tesla V100, connected via NVLINK2, which provides a total bandwidth of 300 GB/s. Each node includes 768 GB DDR4 RAM. Unfortunately we can not generate CPU scalings for more modern hardware due to lacking access to more modern CPU hardware in sufficient numbers.

4.5.3 Selection of the fastest NNS algorithm

In order to use the best available NNS algorithm for our applications, we test all three NNS algorithms previously available in HOOMD-blue on both the CPU and the GPU. We do this by running 100 time steps with one BinAC node or one K80 GPU each for the LDC and the PSA-HIGH benchmarks. We choose these two benchmarks because the LDC benchmark is actually very thin in one dimension and thus represents a special case, and the PSA-HIGH benchmark can be seen as a deputy for the other benchmarks, which do not differ in the arrangement and structure of the particles since for the neighborhood search the property of fluid or solid particles is irrelevant. The results are shown in table 4.4, and point out that the tree algorithm almost always yields the best results. Except for the LDC benchmark on the GPU, the cell algorithms is the better choice. In the following numerical simulations we use the most suitable NNS algorithms, i.e. for the LDC benchmark on the GPU the cell algorithm and otherwise always the tree algorithm.

	CPU		K80	
	LDC	PSA-HIGH	LDC	PSA-HIGH
cell	0.23	0.13	0.39	0.18
stencil	0.23	0.14	0.38	0.19
tree	0.24	0.14	0.35	0.21

Table 4.4: Obtained TPS with the different NNS algorithms for the LDC and PSA-HIGH benchmark.

4.5.4 Evaluation of the improvements

In these experiments we want to investigate whether the modifications described in section 4.4.3 lead to a performance increase. Since the modifications have no direct impact on communication, we limit ourselves to using only a single GPU.

Therefore, we use both an NVIDIA K80 (one of the two GPUs) from the BinAC cluster as reference for older architecture and an NVIDIA V100 from CS-Storm as reference for current hardware.

Here we test our modifications for the LDC and the PSA-HIGH benchmarks because the first one covers the best possible case and the second one is representative for our use cases of CT-scans. Thereby we execute 100 time steps each. Since the LDC benchmark is too large for the memory of one GPU, we reduce it by a factor of 4 so that it has 2.500.000 fluid particles.

We recall that due to the large particle numbers in the LDC test, the NNS radius corresponds to the cut-off radius of the SPH method. Therefore, in this benchmark the NNS algorithm is executed relatively more often than in the other benchmarks. Table 4.5 lists the results for the LDC benchmark and table 4.6 lists the results for the PSA-HIGH benchmark.

		Baseline	Mod. 1	Mod. 2	Pairs
K80	TPS	0.34	0.36	0.40	0.34
	Accel.		1.06	1.18	1.00
V100	TPS	3.27	3.68	4.00	4.83
	Accel.		1.13	1.22	1.48

Table 4.5: TPS and Accelerations for the different improvements for the LDC benchmark.

		Baseline	Mod. 1	Mod. 2	Pairs
K80	TPS	0.16	0.17	0.21	0.17
	Accel.		1.06	1.18	1.06
V100	TPS	1.14	1.22	1.40	3.05
	Accel.		1.07	1.23	2.68

Table 4.6: TPS and Accelerations for the different improvements for the PSA-HIGH benchmark.

Both benchmarks show that modification 1 produces an acceleration of 6% on the K80 for both benchmarks and modification 2 for LDC an acceleration of 18% and for the PSA-HIGH of 13%.

The benchmarks also show that the new approach with the new NNS and the pairs list do not gain in performance in older architectures such as the NVIDIA K80, as Modification 2 achieves the highest accelerations.

If we use the V100 instead, modification 1 and 2 already lead to higher accelerations than on the K80. In addition, the new approach with the pairs list is the clear winner on this architecture with the TPS value for the PSA-HIGH benchmark even being more than twice as high as with modification 2.

To identify where the advantages of this method are rooted in, we look at the individual runtimes of the NNS, force, density and pressure and boundary computations for one time step. It should be noted that the first time step consists of a preparation step and a time step. This corresponds to two iterations of algorithm 4.1 where the NNS algorithm is executed in both iterations. The corresponding runtimes are shown in table 4.7. In the LDC benchmark the accelerations stem from the NNS and the pressure/density calculation, where the NNS times also include all pre-calculations for the neighbor search. The force calculation on the other hand becomes a bit more expensive. This is due to the fact that this benchmark consists mainly of fluid particles, so there is less warp

	LDC				PSA-HIGH			
	NNS	force	den./pre.	bound.	NNS	force	den./pre.	bound.
Mod. 2	357	173	187	2	548	461	574	38
Pairs	180	187	84	2	244	368	247	30

Table 4.7: Walltimes in ms for one iteration on a Tesla V100.

divergence due to different particle interactions. Another reason is that the domain is very homogeneous, so that all particles have the same number of neighbors. Also, the fact that only fluid particles occur in the interior of the domain, and thus no further re-sorting by particle pairs occurs, ensures, that the slow atomic operations occur more frequently in this benchmark.

With the PSA-HIGH benchmark the runtime reductions are even higher. It is mentioned again that in this case the NNS is not performed in every time step, so that in table 4.6 we see only one factor which corresponds to the runtime reduction in the force computation.

As in section 4.4.2, we considered again the warp divergence of the three main SPH components force computation, pressure and density computation and the noslip boundary condition computation, while using the presented algorithm this time, which creates a pairs list. The incidence of warp divergence is measured using NVIDIA Visual Profiler, where we consider the first call in the time loop at a time. The resulting numbers are shown in table 4.8, where each row describes one warp divergence in the corresponding kernel. Warp divergences occur in up to 70-80 % of the kernels, which is still high and partly even higher than with the Verlet list approach (cf. table 4.2), but there are fewer warp divergences per kernel. Only the force calculation has two warp divergences now instead of one. Furthermore, the warp divergences in this case are all atomic operations (or even warp aggregated atomics) which are no longer as slow on current architectures. In particular, the warp divergence caused by different numbers of neighbors, which contains a lot of instructions and is therefore very expensive, could be eliminated.

Kernel	force eq. (4.37)	dens.+pre. eqs. (4.16)+(4.28)	noslip eqs. (4.30)+(4.31)
Divergent Execution in %	75.7	75.8	84.6
	52.6	39.7	54.3

Table 4.8: Warp divergence using the modified NNS algorithm, single GPU K40c.

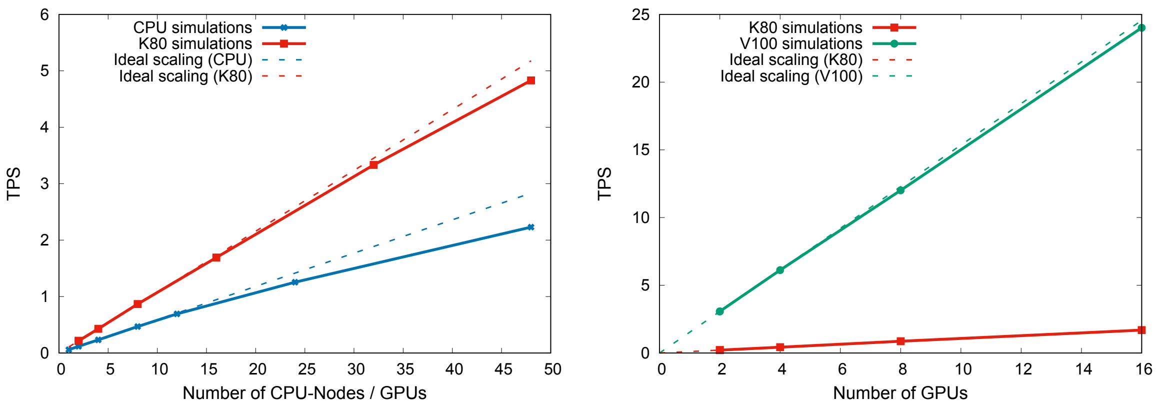
4.5.5 Strong scalability

We recall that HOOMD-blue, similar to many particle codes, measures performance in TPS, where a higher value indicates better performance. Furthermore, we adopt this metric, and additionally derive the parallel efficiency E_n (normalized to the granularity of nodes n rather than cores/processes/GPUs) and the GPU acceleration factor S as follows:

$$E_n = \frac{\text{TPS}_n/\text{TPS}_1}{\#\text{Compute\_units}} \qquad S = \frac{\text{TPS}_{\text{GPU}}}{\text{TPS}_{\text{CPU}}}$$

Here *#Compute_units* describes a CPU node for the CPU runs, and usually a GPU for the GPU runs. However, we always set *#Compute_units* to the smallest possible number of GPUs, so if the benchmark is too big for the memory of one GPU, we start with two. These normalizations are slightly nonstandard, but we consider them much more meaningful on the application level than the classical textbook definitions of efficiency and speedup.

For each benchmark scenario, we report the TPS metric for 100 time steps, and perform file I/O only at the very beginning (to load the geometry etc.) and the end (to save the simulation result). In each case, we only use the most efficient variant for the respective platform, i.e., Modification 2 with the HOOMD-blue NNS algorithm is used on the BinAC cluster, and we use the new variant with the pairs list on CS-Storm. We use up to 48 GPUs of type K80 for this purpose, because for more GPUs the number of particles is too small to saturate the GPUs with enough work. This corresponds to exactly 12 nodes in the BinAc cluster. We always mean 24 K80 cards when we talk about 48 K80 GPUs, because one K80 card contains two GPUs. Both GPUs in a K80 have 12 GB of memory. A V100 GPU, as included in the CS-Storm cluster, has 32 GB of memory, so 18 V100 GPUs provide the same amount of memory as 48 K80 GPUs. To use only whole nodes, we use up to 16 V100 GPUs for this scaling investigation. Figures 4.15–4.18 depict the results of these strong scaling tests, whereas figure (a) always shows the CPU and the K80 scaling and figure (b) compares the K80 and the V100 simulations. Precise numbers can be found in the in tables 4.9–4.12.



(a) (b)

Figure 4.15: Strong scalability – LDC benchmark. (a) TPS for simulations on CPU and K80, (b) TPS for simulations on K80 and V100.

For the LDC test, we observe almost perfect strong scalability for the CPU and GPU implementation. This is expected, because almost all particles represent the fluid phase, and their distribution is very homogeneous with respect to the spatial domain decomposition. The experiments for even larger node counts emphasize this, as efficiency degrades very slowly (see table 4.9). Thus the parallel efficiency E for 12 CPU nodes is 98 %, that for 416 K80 GPUs 98 % and that for 16 V100 GPUs 98 %. Only beyond 24 CPU nodes and 48 K80 GPUs does the parallel efficiency decrease to 89 % and 93 % respectively. Here it is important to note that the pairs list based neighbor search is used for the simulation on the V100. Although a V100 with 5,120 CUDA cores has more than twice as much as a GPU in a K80 card (with 2496 cores), there are many more neighbor pairs than particles, so the pairs list approach, where a thread can be associated with a neighbor pair, can hide latencies better.

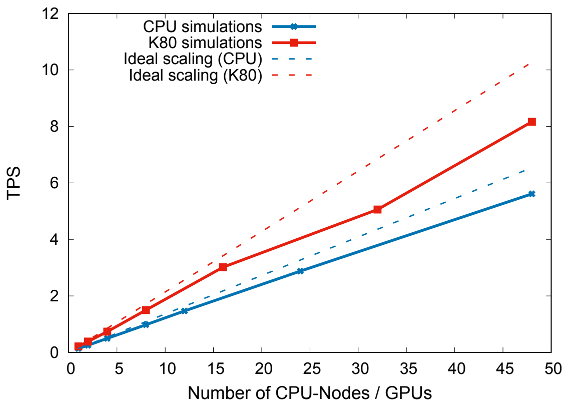
#CPU nodes /	CPU nodes		GPU K80			GPU V100		
#GPUs	TPS	E	TPS	E	Accel.	TPS	E	Accel.
1	0.059		N/A			N/A		
2	0.119	1.01	0.216		1.82	3.07		25.80
4	0.233	0.99	0.428	0.99	1.84	6.12	1.00	26.27
8	0.469	0.99	0.867	1.00	1.85	12.01	0.98	25.61
12	0.693	0.98						
16			1.692	0.98		24.01	0.98	
24	1.254	0.89						
32			3.333	0.96				
48	2.517	0.89	4.830	0.93	1.92			

Table 4.9: Strong scalability – LDC benchmark.

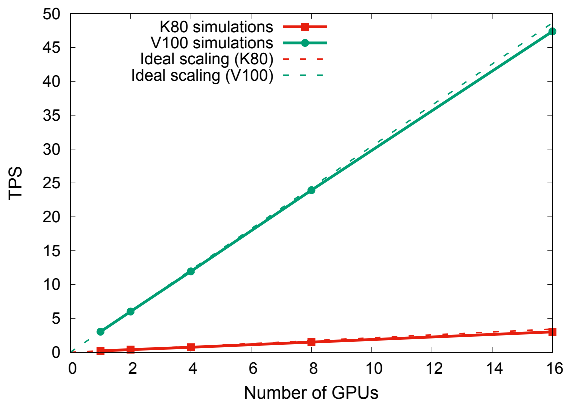
The PSA-HIGH scenario scales only slightly worse, maintaining 90 % and 88 % parallel efficiency for 12 CPU nodes and 16 K80 GPUs respectively, see table 4.10 for the exact numbers. If the number of GPUs is increased and 48 K80 GPUs are used, the parallel efficiency is still 79 %. The parallel efficiency for 48 CPU nodes is with 86 % even better. This small degradation compared to the previous test case is expected, because the overall number of particles is cut in half compared to the previous benchmark, (see table 4.3), and also the discussion of the next experiment. In addition, the ratio of fluid to solid particles is slightly less favorable. Also the numbers for the runs on the V100 are comparable to the K80 ones. As in the LDC test we can see that the parallel efficiency decreases slower than for the simulations on the K80.

#CPU nodes /	CPU nodes		GPU K80			GPU V100		
#GPUs	TPS	E	TPS	E	Accel.	TPS	E	Accel.
1	0.136		0.214		1.57	3.04		22.35
2	0.255	0.94	0.387	0.90	1.51	6.01	0.99	23.57
4	0.496	0.91	0.738	0.86	1.49	11.94	0.98	24.07
8	0.981	0.90	1.501	0.88	1.53	23.93	0.98	24.39
12	1.468	0.90						
16			3.018	0.88		47.38	0.97	
24	2.878	0.88						
32			5.057	0.74				
48	5.612	0.86	8.166	0.79				

Table 4.10: Strong scalability – PSA-HIGH benchmark.



(a)



(b)

Figure 4.16: Strong scalability – PSA-HIGH benchmark. (a) TPS for simulations on CPU and K80, (b) TPS for simulations on K80 and V100.

As expected, the PSA-LOW scenario scales slightly worse (especially on the CPU), as there are much more solid than fluid particles: We simulate fluid flow in a domain that is specified by spatially stationary solid particles. Hence, the neighbor list update of the fluid particles requires mostly local communication (the fluid particles move more or less at the same speed), whereas the neighbor list update for fluid-solid interaction increases the amount of non-local communications, simply because fluid particles retain their 'fluid' neighbors but vary their 'solid' neighbors. For low node counts, we observe acceptable efficiencies of 90 % on 12 BinAC nodes (288 cores). If we increase to 48 CPU nodes the parallel efficiency decreases to 48 %. When using few GPUs, the latencies can still be hidden very well, and the parallel efficiency is good for 8 K80 GPUs with $E = 0.96$ and nearly perfect for up to 8 V100 GPUs with $E = 0.98$. But if we increase the number to 48 GPUs (K80), the parallel efficiency decreases to 73 %. The noticeable loss of parallel efficiency for more than 32 GPUs (8 GPU nodes) is explained by the number of fluid particles per GPU: As soon as the number of particles drops below the maximum number of simultaneous CUDA threads per GPU, latency hiding becomes impossible in general. Also for the pairs list approach on the V100, the behavior is similar to that on the K80.

#CPU nodes /	CPU nodes		GPU K80			GPU V100		
#GPUs	TPS	E	TPS	E	Accel.	TPS	E	Accel.
1	0.349		0.338		0.97	6.84		19.60
2	0.647	0.93	0.672	0.99	1.03	13.57	0.99	20.97
4	1.128	0.81	1.295	0.96	1.15	26.99	0.99	23.93
8	2.033	0.73	2.596	0.96	1.28	53.87	0.98	26.50
12	2.891	0.69						
16			4.931	0.91		100.01	0.91	
24	5.061	0.60						
32			9.117	0.84				
48	8.081	0.48	11.767	0.73	1.46			

Table 4.11: Strong scalability – PSA-LOW benchmark.

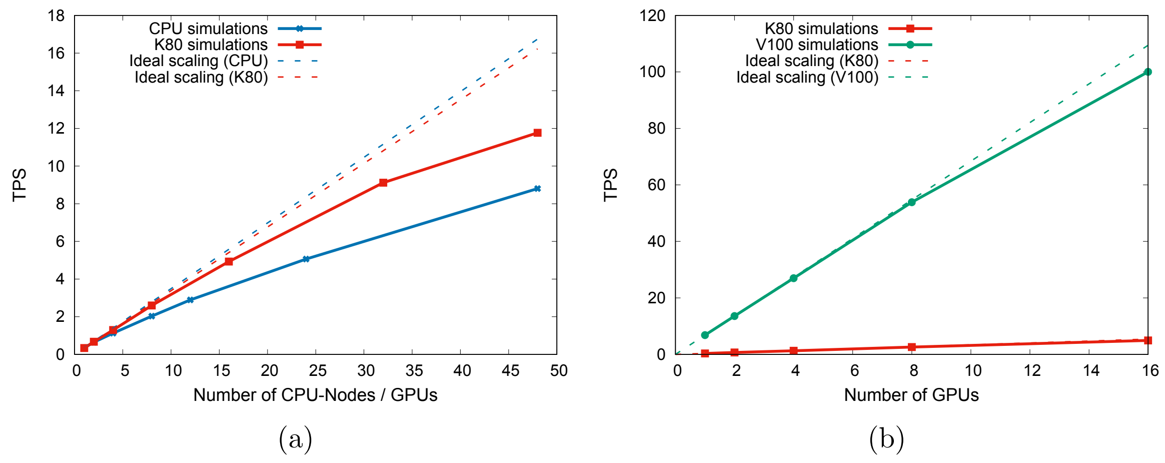


Figure 4.17: Strong scalability – PSA-LOW benchmark. (a) TPS for simulations on CPU and K80, (b) TPS for simulations on K80 and V100.

Finally, the VC benchmark (figure 4.18) scales only moderately well on CPUs, with 58 % parallel efficiency for 12 CPU nodes and 52 % for 48 CPU nodes, or 54 % parallel efficiency for 16 K80 GPUs and 49 % parallel efficiency for 48 K80 GPU, and 61 % parallel efficiency for 16 V100 GPUs. Even though the solid-to-fluid ratio is more favorable than in the previous case, the domain decomposition leads to strongly varying neighbor counts, and thus scaling detriments because of the irregularity of the problem: Technically, one GPU is associated with one MPI rank, while one BinAC node (12+12 cores) executes 24 MPI ranks, which leads to more and smaller MPI messages, since the domain of each rank has more neighbors, but the interfaces to the neighbors become smaller. Load balancing (see section 4.4.1) yields a small improvement as we have shown in [128].

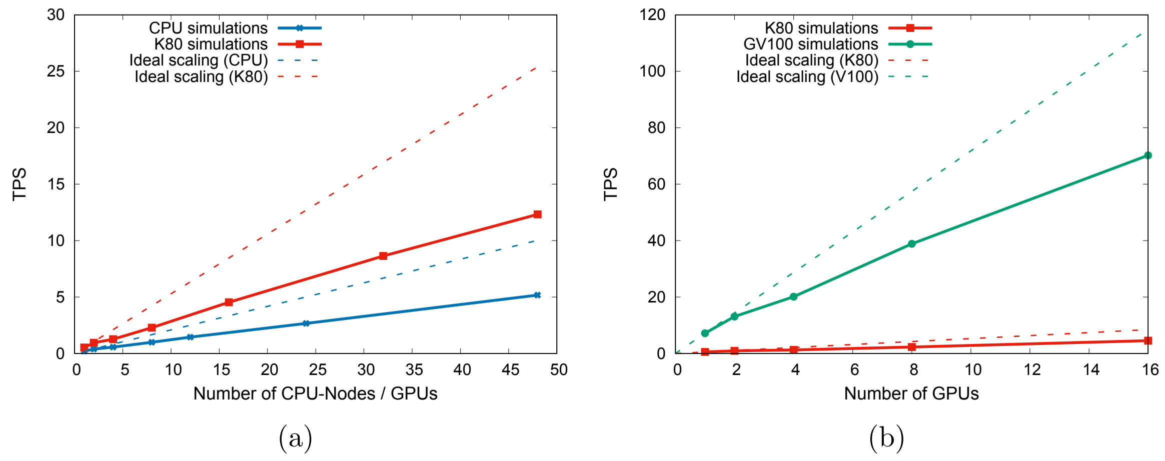


Figure 4.18: Strong scalability – VC benchmark. (a) TPS for simulations on CPU and K80, (b) TPS for simulations on K80 and V100.

Also the accelerations GPU vs. CPU can be taken from the tables 4.9–4.12. Both accelerations are accelerations compared to the CPU runs. The highest accelerations are obtained for the two cases which consist mainly of fluid particles and are only bounded by solid particles and interior domain consists only of fluid particles. This is because computations of fluid fluid interactions are easier to parallelize (without causing warp

#CPU nodes / #GPUs	CPU nodes TPS	E	GPU K80 TPS	E	Accel.	GPU V100 TPS	E	Accel.
1	0.209		0.529		2.53	7.18		34.35
2	0.393	0.94	0.947	0.90	2.41	13.98	0.97	35.57
4	0.567	0.68	1.264	0.60	2.23	20.12	0.70	35.49
8	0.990	0.59	2.284	0.54	2.31	38.87	0.68	39.26
12	1.452	0.58						
16			4.535	0.54		70.21	0.61	
24	2.668	0.53						
32			8.636	0.51				
48	5.170	0.52	12.325	0.49	2.38			

Table 4.12: Strong scalability – VC benchmark.

divergences) than computations of fluid solid interactions. For the VC benchmark the accelerations are between 2.23 and 2.51 for the K80. For the LDC benchmark, the K80 achieves accelerations around 1.8. The V100 achieves accelerations between 19.60 and 39.26 which seems high, but is a bit unfair because of comparing against a CPU architecture that is one generation older. Furthermore, the V100 yields between ∼13 (LDC benchmark) and ∼24 (PSA-LOW) times faster results than the K80. Slightly lower are the accelerations for the PSA-HIGH benchmark, which are between 1.49 and 1.57 for the K80. However, due to the better parallel efficiency of the GPU implementation, the acceleration for more CPU nodes or GPUs increases. The same behavior can be seen in the PSA-LOW benchmark, where in this case a GPU is slower than a CPU node. For the K80 the accelerations are only between 0.97 and 1.46.

4.5.6 Weak scalability

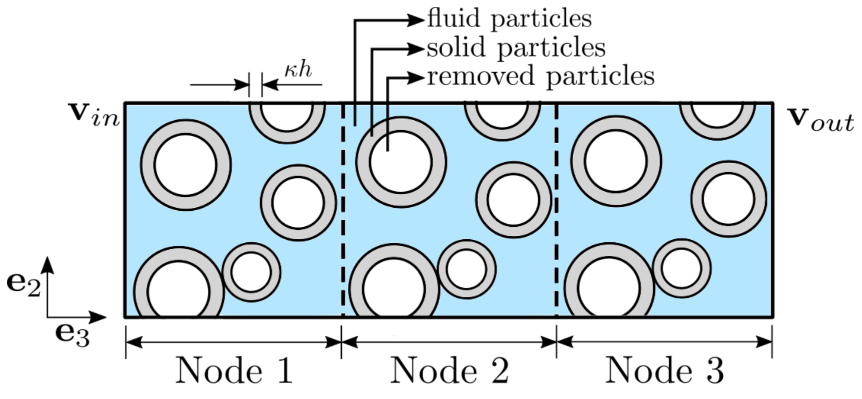


Figure 4.19: Domain setup for weak scaling study.

Next we investigate weak scalability. Figure 4.19 shows the setup for the weak scaling analysis. The domain is periodically replicated in the $\mathbf{e}_3$-direction, in such a way that after the domain decomposition, each node contains the same amount of fluid and solid particles (which are later denoted with the index $\mathfrak{f}$ and $\mathfrak{s}$, respectively). Additionally, the solid particles that have no interaction with the fluid particles ($|\mathbf{x}_\mathfrak{s} - \mathbf{x}_\mathfrak{f}| > \kappa h$) are removed from the domain. Periodic boundary conditions are set in $\mathbf{e}_3$-direction, i.e., $\mathbf{v}_{in} = \mathbf{v}_{out}$. This 'application weak scaling' may lead to slight variations of overall particle counts, as the number of particles associated with periodic boundary conditions remains more or less constant, while the overall number of particles increases proportionally. The results of the weak scaling test series of the two Packed-Spheres benchmarks with high and low porosity distribution are shown in figures 4.20 and 4.21 depict.

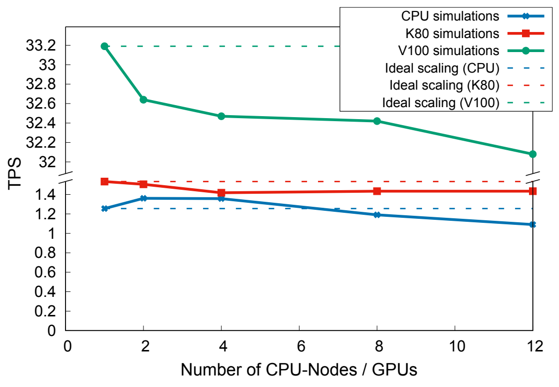


Figure 4.20: Weak scalability – PSA-LOW scenario.

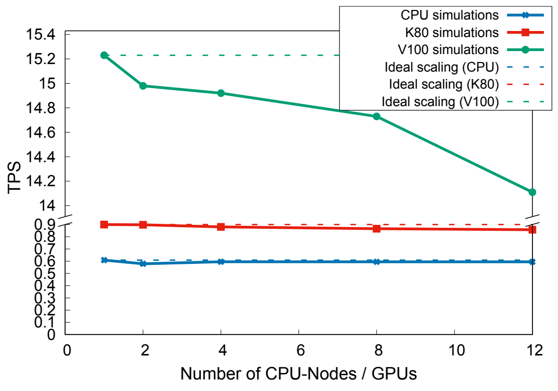


Figure 4.21: Weak scalability – PSA-HIGH scenario.

For the K80 GPU implementation we clearly observe perfect weak scaling when factoring out the specific details of the BinAC architecture: If we use two GPUs, we stay in the same card (with distinct physical memory), extending to four GPUs still does not involve communication over the network since two K80 GPUs are in one node. Going beyond four GPUs involve Infiniband communication and yields almost perfect weak scaling behavior. Since we start with a GPU and a CPU node we can better compare the CPU and GPU implementation, since a CPU node uses only a single shared memory, as does a GPU. This means that both one CPU node and one GPU only need internal communication and memory has to be transferred when moving to 2 CPU nodes or 2 GPUs.

A similar reasoning holds for the CPU runs, at least in the high porosity scenario which comprises much more uniformly-behaving fluid particles than solid particles. The loss of scaling on CPUs compared to GPUs for the PSA-LOW benchmark (cf. figures 4.21 and 4.20) is mostly determined by the different initial spatial domain decompositions: One GPU is associated with one MPI rank, while 12+12 BinAC execute one MPI rank each. Despite all shared memory optimizations of the underlying OpenMPI library, the communication pattern of the CPU-only implementation is substantially different than on GPUs with OpenMPI, and the observed scaling weakness takes place on CPUs much earlier than on GPUs. This is also the reason why in the PSA-LOW scenario the TPS value increases when going from one to two nodes, which is also reproducible on another system [128]. If we look at the TPS values for the weak scaling of the V100 GPUs in figure 4.20 and 4.21, the graph of weak scaling looks worse in this case. However, if we consider the percentage loss when increasing the number of GPUs, it is almost the same as when increasing the number of K80 GPUs. Just like before with the scaling for the K80 GPUs,

#CPU nodes / #GPUs	CPU nodes TPS	GPU K80 TPS	GPU K80 Accel.	GPU V100 TPS	GPU V100 Accel.
1	0.609	0.900	1.48	15.23	25.01
2	0.579	0.898	1.55	14.98	25.87
4	0.596	0.881	1.48	14.92	25.03
8	0.595	0.866	1.46	14.73	24.76
12	0.595	0.858	1.44	14.11	23.14

Table 4.13: Weak scalability – PSA-HIGH benchmark.

the decrease can be explained by the architecture. In the step from one to two GPUs the NVLINK connection is applied. If we increase the number of cards to four and eight there is only little loss of speed, all GPUs are connected via NVLINK, but each GPU has to communicate with more than one other GPU. A further speed reduction occurs when switching from 8 to 12 GPUs. Here the communication between two nodes is additionally required, which is realized via an Infiniband connection.

The exact numbers can be found in tables 4.13 and 4.14.

#CPU nodes / #GPUs	CPU nodes TPS	GPU K80 TPS	GPU K80 Accel.	GPU V100 TPS	GPU V100 Accel.
1	1.255	1.532	1.22	33.19	26.44
2	1.361	1.505	1.11	32.64	23.98
4	1.357	1.418	1.04	32.57	24.00
8	1.191	1.434	1.20	32.42	27.22
12	1.091	1.434	1.31	32.08	29.40

Table 4.14: Weak scalability – PSA-LOW benchmark.

4.6 Conclusions

In this chapter we have considered the particle-based simulation of flow in porous media and presented implementation aspects for the SPH method. This implementation is challenging for the GPU because the SPH method is a mesh-free method and therefore leads to an unstructured data structure and data access.
However, this method is also computationally intensive so parallel computing is essential and GPU computing is helpful. After we described the challenges in section 4.4.2, we have presented possible improvements in the following section 4.4.3.
Therefore we used following advanced GPU programming techniques:

- We have transformed the Array of Structures approach into a Structures of Arrays. This is better suited for the architecture and memory access of a GPU. This leads to less cache misses resulting in less latencies and idle times.

- By smartly fusing different kernel functions it is possible to avoid latencies that occur when launching the kernel functions. A positive side effect is often that data can be reused, which leads to less cache misses. Nevertheless you have to take care not to add additional synchronization barriers.

- Especially on older GPUs atomic operations are very complex, which is why many state of the art NNS algorithms do not use them. But on current architectures they are often not that slow anymore. Warp-aggregated atomics also offer the possibility to hide the disadvantages of atomic operations. We also use warp-aggregated atomics in this variant in the SPH kernels, because we use a pairs list instead of the Verlet list. This transition from an atomic avoidance strategy to an approach where less operations have to be performed but more atomic operations occur, leads to considerable improvements on current hardware accelerations.

These improvements have been evaluated in section 4.5.4 and we have seen that the basic improvements on the old K80 architecture have led to accelerations, but even better accelerations on current hardware. The approach presented in section 4.4.3 of modifying the NNS algorithm and using a pairs list instead of the Verlet list does not lead to improvements on older architectures like the K80. On new architectures, however, we have seen a significant acceleration.
Furthermore, we have shown in sections 4.5.5 and 4.5.6 that our implementation scales well on the CPU and GPU in both the strong and weak sense.

5

Seismic waveform modeling and inversion

In this chapter, we consider the modeling of wave propagation in anelastic materials and the waveform inversion. We start in section 5.2 to derive the mathematical model of wave propagation in anelastic materials. Since the standard modeling approach is not suitable for the inversion of all physical parameters, we then modify this approach in section 5.3.1. Here we also modify the optimization problem which determines the relaxation parameters for the underlying rheological model for a given Q factor. Based on these modifications we analyze in section 5.3.3 the accuracy of the resulting Q factor approximation and compare it with the standard approach. In section 5.4 we introduce the waveform inversion, define the minimization problem and introduce possible regularization procedures. The gradient of the minimization problem, which is needed to solve the minimization problem, is calculated using the adjoint state method, which is described in section 5.4.3. This method requires the solution of the adjoint problem, which is derived in section 5.4.4. Afterwards, we show how to compute the resulting gradient in section 5.4.5. The previously derived models and methodologies are used in section 5.5 to describe an implementation realization. Our presented methods and modifications are studied numerically in section 5.6. On the one hand, we investigate the influence of attenuation for the adjoint simulation to verify whether attenuation is necessary in the adjoint simulation. Furthermore, we compare different regularization methods, since our inverse problems are ill-posed and the addition of attenuation might have an influence on regularization. In addition, we present a modified inversion method that can provide faster and better inversion results. Since the modeling of wave propagation and inversion has become more complex due to the extension of the method with the possibility of inverting the Q factors, we need efficient implementations and powerful hardware in addition to efficient solver methods. Moreover, our main application is in the area of geophysics where the problems encountered in practice are very large (e.g., high number of sources and receivers, moderately large areas, and inversion problems are costly anyway), so the use of parallel computers is essential and the use of GPUs helpful. Therefore, our algorithm supports CPU and GPU implementations and is suitable for small clusters and supercomputers. We investigate our implementation *concerning strong and weak scalability in section 5.6.*

Since we use the finite differences method to discretize the model equations, which is a very structured method and thus a method that can be parallelized well, this chapter does not focus on the implementation details and possible improvements, but on the mathematical improvement of the model and the reduction of the modeling error.

5.1 Motivation

Seismic waveform inversion is an imaging technology that is used in geophysics but also has other applications. For instance, it is used in the oil industry to find hidden oil reservoirs or in medicine to obtain an image of the human brain without using X-ray tomography [65]. The basic idea of this method is to minimize the mismatch between synthetic and objective data. For example, the objective data may result from a physical experiment, or in the simulation context, from a solved forward problem with the desired medium. These objective data are also called seismograms. However, this information does not allow direct conclusions to be made regarding the geophysical properties of the Earth's structure. To obtain an image of the Earth or other objects an inversion is necessary. The synthetic data are the data generated by modeling the forward problem using the reconstructed medium given by the method, which is iteratively improved. This minimization problem is usually very non-linear and represents an ill-posed inverse problem. The searched medium is described, e.g., by pressure and shear velocities or Lamé parameters in the case of geological media, or other properties that describes a material.

The material to be reconstructed can be different, such as acoustic or elastic material. However, real material of the Earth is anelastic. In this thesis, we therefore consider viscoelastic media which approximates the behavior of real Earth material. The viscous attenuation can have a strong influence on the propagation of seismic waves but is rarely taken into account in the full waveform inversion (FWI). Full waveform inversion is a popular inversion technique in geophysics. It is essentially based on the adjoint theory developed, among others, by Lailly [92], Tarantola and Valette [147] and Crase et al. [47]. Furthermore, such an approach is also necessary to reconstruct the damping parameters [52]. To solve the minimization problem of the inversion, commonly iterative techniques are used, like Gauß-Newton or non-linear conjugate gradient methods, to update the initial model by perturbing it gradually through these cost function gradients. Quasi-Newton methods like the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm and its optimized l-BFGS version [119, 145] are used in the time domain to invert envelope measurements or time arrivals [29, 31, 87, 103, 146, 156] or full waveforms [22, 115, 162] at the global or regional scales.

Due to the ill-posedness of inverse problems, regularization procedures are helpful to make the procedures more robust against, e.g., disturbed input data [4, 58]. The quality of regularization procedures depends on the structure of the model and the equations of the forward model. Therefore we want to investigate the influence of established regularization methods for acoustic and elastic problems on the viscoelastic problems.

In practice, such applications become very large and inverse problems are additionally computationally challenging. Furthermore, the computational effort is further increased by the use of viscoelastic equations in both forward and adjoint modeling. An efficient implementation is, therefore, necessary to obtain the simulation results in the shortest possible time despite the increased computational effort. In addition, GPUs are suitable for further accelerating the simulations due to the possible choice of structured discretization methods.

5.2 Mathematical model

In this section, we derive a mathematical model for wave propagation in anelastic media. For this purpose, we start with the general wave equation for any relaxation function. Then, we derive a relaxation function that describes the behavior in anelastic material. Therefore, we derive a corresponding rheological model.

5.2.1 Wave equation

The conservation of linear momentum implies

$$\rho \partial_{tt} u = \nabla \cdot \boldsymbol{\sigma} + f, \tag{5.1}$$

where $u = (u_x \; u_y)^\top$ is the displacement vector, ρ is the mass density and $f = (f_x \; f_y)^\top$ are the body forces. A general relation between the components of the stress tensor $\boldsymbol{\sigma}$ and the components of the strain tensor $\boldsymbol{\varepsilon}$ is given by

$$\sigma_{ij}(x,t) = \Psi_{ijkl}(x,t) * \dot{\varepsilon}_{kl}(x,t). \tag{5.2}$$

The components of the strain tensor are defined as

$$\boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} & \varepsilon_{xy} \\ \varepsilon_{yx} & \varepsilon_{yy} \end{pmatrix} = \begin{pmatrix} \partial_x u_x & \frac{1}{2}(\partial_x u_y + \partial_y u_x) \\ \frac{1}{2}(\partial_x u_y + \partial_y u_x) & \partial_y u_y \end{pmatrix}$$

and the the components of the stress tensor are named as

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{xy} & \sigma_{yy} \end{pmatrix}.$$

The most general isotropic fourth-order tensor (in 2D) is

$$\Psi_{ijkl}(x,t) = \frac{1}{2}\left[\Psi_1(x,t) - \Psi_2(x,t)\right]\delta_{ij}\delta_{kl} + \frac{1}{2}\left[\Psi_2(x,t)\right](\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}), \tag{5.3}$$

where $\Psi_1(x,t)$ and $\Psi_2(x,t)$ are relaxation functions and δ_{ij} is the Kronecker delta. $\Psi_1(x,t)$ describes dilatational deformations and $\Psi_2(x,t)$ describes shear deformations.

By selecting the relaxation functions, the propagation properties and/or material properties are described. Therefore we need a relaxation function that describes the properties of anelastic material.

5.2.2 Zener model

Our objective is to describe wave propagation in real Earth material. Therefore we derive a rheological model, that represents the properties of anelastic material. Rheology is the branch of physics that studies the way in which materials deform or flow, in response to applied forces or stresses. The material properties that govern the specific way in which these deformation or flow behaviors occur are called rheological properties and the model that describes this behavior is called the rheological model. We use the so-called Zener model which we derive in the following, based on [40, 111].

Anelastic materials Elastic media can be described very simply, but they have no loss of internal energy, whereas real materials do not behave in this way. Materials that behave differently from the elastic media are called anelastic. The deviation from the elastic behavior of materials is anelasticity. The simplest rheological model of an anelastic material is a linear viscoelastic body that combines two extreme behaviors, linear elasticity, and linear viscosity. A material is linearly viscoelastic if the stress tensor is linearly related to the strain tensor, and the strain response to a linear combination of applied stresses is the same linear combination of strain responses to individually applied stresses.
In the following, we present a rheological model for a linear elastic and a linear viscous material and finally combine them.

Linear elastic body The linear elastic body, also known as Hooke body, represents the behavior of a perfectly elastic (lossless) solid material. In other words, stress is proportional to strain:

$$\sigma(t) = M \cdot \varepsilon(t) \tag{5.4}$$

Here $\sigma(t)$ is the stress as a function of time $t, \varepsilon(t)$ the strain, and M the time-independent elastic modulus. An application of a load yields an instantaneous deformation. A removal of the load yields instantaneous and total recovery. The Hooke body does not have a memory, that means stress at a given time only depends on the deformation at the same time. The Hooke body consists of only a single elastic spring and is shown in figure 5.1. The strain-time diagram for constant stress applied at time t_0 and removed at time t_1 is shown in figure 5.2, right, the stress-strain diagrams in figure 5.2, center and left. To

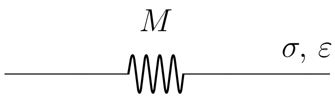


Figure 5.1: Hooke body.

be able to analyze the model in both the time and frequency domains, we also want to

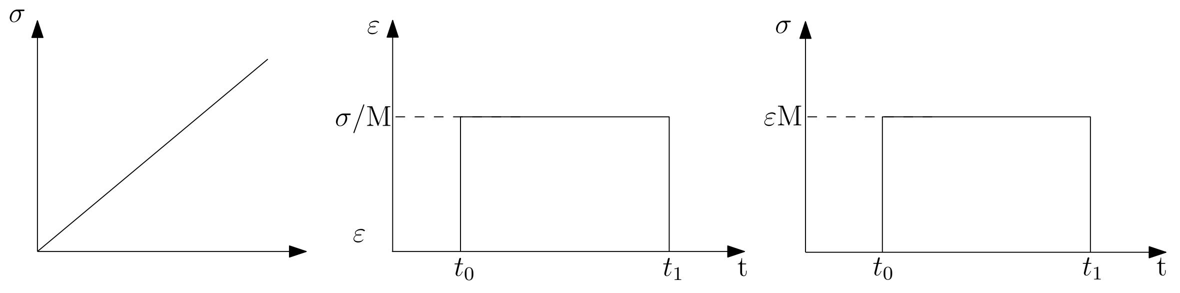


Figure 5.2: left: Stress-strain diagram, right: Stress-time diagram for a constant stress.

describe the model in the frequency domain. Therefore, to transform equation (5.4) into the frequency domain we use the Fourier transformation and get

$$\sigma(\omega) = M \cdot \varepsilon(\omega). \tag{5.5}$$

Linear viscous body Linear viscous body, also known as Stokes body, represents the other extreme behavior in the variety of linear rheological bodies, the behavior of the viscous fluid. That means stress is proportional to strain rate:

$$\sigma(t) = \eta \cdot \dot{\varepsilon}(t) \tag{5.6}$$

Here η is the time-independent viscosity. An application of a load yields non-instantaneous linearly increasing deformation. Removal of the load does not yield the removal of deformation, so there is no recovery. In contrast to the Hookes body, the Stokes body has extreme memory. The Stokes body consists of a dashpot and is shown in figure 5.3. The strain-time diagram for constant stress applied at time t_0 and removed at time t_1 is shown in figure 5.4 (right), the stress strain-rate diagrams in figure 5.4 (center and left). Again

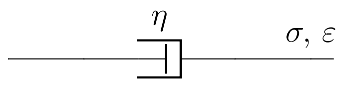


Figure 5.3: Stokes body.

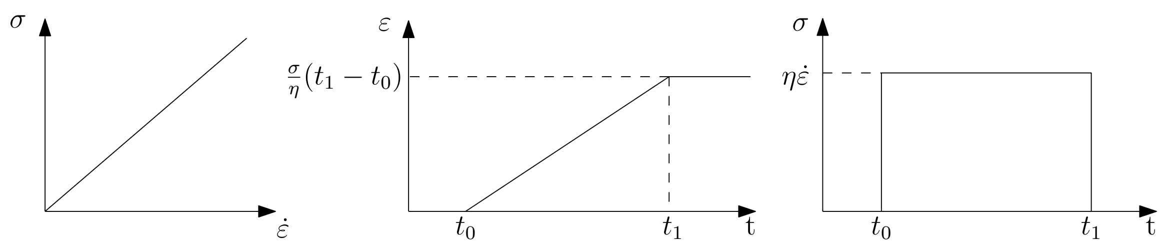


Figure 5.4: left: Stress-strain-rate diagram, right: Strain-time diagram for a constant stress.

we want to describe the model also in the frequency domain. To transform equation (5.6) into the frequency domain, we use the Fourier transformation and obtain

$$\sigma(\omega) = i\omega\eta \cdot \varepsilon(\omega). \tag{5.7}$$

Stress-strain relation in viscoelastic medium Many real materials combine the behaviors of both, elastic solids and viscous fluids. As a consequence, these materials remember their past, or in other words, the stress-strain relation also depends on time. We can approximate such behavior using viscoelastic models of a medium. Before we present a rheological model as a combination of Hooke and Stokes bodies, we first determine the relation of stress and strain for a viscoelastic medium, as well as the stress function and modulus function. For a linear isotropic viscoelastic material, the stress-strain relation is given by the Boltzmann superposition and causality principle. In a simple scalar notation, it is given by

$$\sigma(t) = \int_{-\infty}^{t} \Psi(t-\tau)\dot{\varepsilon}(\tau)\mathrm{d}\tau \tag{5.8}$$

(the convolution of Ψ with $\dot{\varepsilon}$), where $\sigma(t)$ is stress, $\dot{\varepsilon}(t)$ time derivative of strain, and $\Psi(t)$ stress relaxation function. According to equation (5.8), the stress at a given time t is determined by the entire history of the strain until time t. The upper integration limit ensures the causality. We use the symbol '$*$' for the convolution so that equation (5.8) then can be written as

$$\sigma(t) = \Psi(t) * \dot{\varepsilon}(t). \tag{5.9}$$

Due to properties of convolutions we can transfer the derivative to the relaxation function:

$$\sigma(t) = \dot{\Psi}(t) * \varepsilon(t) \tag{5.10}$$

Since $\Psi(t)$ is the stress response to a unit step function in strain, its time derivative is the stress response to the Dirac δ-function in strain

$$M(t) = \dot{\Psi}(t). \tag{5.11}$$

We use the identity in equation (5.11) and rewrite equation (5.10) as

$$\sigma(t) = M(t) * \varepsilon(t). \tag{5.12}$$

Now we can compare equation (5.12) with equation (5.4). We see that the stress-strain relation for the elastic body is a simple linear relation with a constant elastic modulus and the stress-strain relation for the viscoelastic body has a convolutory form as a consequence of the time-dependent modulus $M(t)$. We transform equation (5.12) into the frequency domain by an application of the Fourier transform $\mathcal{F}$ and obtain

$$\sigma(\omega) = M(\omega) \cdot \varepsilon(\omega), \tag{5.13}$$

where

$$M(\omega) = \mathcal{F}\{M(t)\} = \mathcal{F}\{\dot{\Psi}(t)\} \tag{5.14}$$

is the complex, frequency-dependent viscoelastic modulus. Again we compare equation (5.13) with the stress-strain relation for the linear elastic body, i.e., equation (5.5) and see that they are the same except from the point that the modulus function is frequency dependent in the viscoelastic body. An application of the inverse Fourier transformation to equation (5.14) results in

$$\dot{\Psi}(t) = \mathcal{F}^{-1}\{M(\omega)\} \tag{5.15}$$

and, due to properties of the Fourier transformation,

$$\Psi(t) = \mathcal{F}^{-1}\left\{\frac{M(\omega)}{i\omega}\right\}. \tag{5.16}$$

Equation (5.13) indicates that the incorporation of the linear viscoelasticity and consequently attenuation into the frequency-domain computations is much easier than those in the time-domain computations – real frequency-independent moduli are simply replaced by complex, frequency-dependent quantities. If we use equation (5.10) we can rewrite the time derivative of the stress as

$$\dot{\sigma} = \dot{\Psi}(t) * \dot{\varepsilon} \tag{5.17}$$

or, if we use equation (5.11) as

$$\dot{\sigma} = M(t) * \dot{\varepsilon}. \tag{5.18}$$

In practice, it is often useful to know the relaxation function and the modulus function at time zero and after infinite time or at infinite frequency, for example to determine the relaxed and unrelaxed moduli. Therefore, we consider equation (5.14):

$$M(\omega) = \mathcal{F}\left\{\dot{\Psi}(t)\right\} = \int_{-\infty}^{\infty} \dot{\Psi}(t)\exp(-i\omega t)\mathrm{d}t \tag{5.19}$$

It can be shown [40] that this can be rewritten to

$$M(\omega) = \Psi(\infty) + i\omega\int_{0}^{\infty}\left[\Psi(t) - \Psi(\infty)\right]\exp(-i\omega t)\mathrm{d}t$$

and with this formulation it follows that

$$M(\omega = 0) = \Psi(t = \infty). \tag{5.20}$$

Furthermore, we know that $i\omega\mathcal{F}\{\phi(t)\} = \phi(t=0)$ for $\omega \to \infty$ which leads to

$$M(\omega = \infty) = \Psi(t = 0). \tag{5.21}$$

Using the relations (5.20) and (5.21), we can define the following characteristics: An instantaneous elastic response of the viscoelastic material is given by the so-called unrelaxed modulus M_U

$$M_U = \lim_{t\to 0} \Psi(t),$$

a long-term equilibrium response is given by the relaxed modulus M_R

$$M_R = \lim_{t\to\infty} \Psi(t).$$

In the frequency domain, the relaxed and unrelaxed moduli are given by

$$M_U = \lim_{\omega\to\infty} M(\omega) \text{ and } M_R = \lim_{\omega\to 0} M(\omega). \tag{5.22}$$

Furthermore, we need a quantifier that describes the attenuation property of the viscoelastic material. One such quantifier is the so-called Q factor. Given the viscoelastic modulus, the quality factor $Q(\omega)$ is

$$Q(\omega) = \Re M(\omega)/\Im M(\omega). \tag{5.23}$$

where $\Re$ describes the real part and $\Im$ the imaginary part.

It can be shown that $1/Q(\omega)$ is a measure of internal friction in a linear viscoelastic body. It is obvious that numerical integration of the stress-strain relation (5.8) is practically intractable due to the large computer time and memory requirements. This led many modelers to incorporate only oversimplified $Q(\omega)$ laws in the time-domain computations.

Rules for linear rheological models After having described the stress-strain correlation in the viscoelastic medium, we can combine the Hooke body, which describes the linear elastic behavior, and the Stokes body, which describes the linear viscous body, to a rheological model that describes the viscoelastic behavior. Models that quite well approximate rheological properties and behavior of the real Earth's material can be constructed by connecting the simplest rheological elements, Hooke and Stokes elements, in parallel or series. The properties of the models can be analyzed in the time and frequency domains. There are relatively simple rules in both domains that allow obtaining mathematical representations of the models as we have seen above. With the help of these rules, we derive a model for viscoelastic material in the following section.

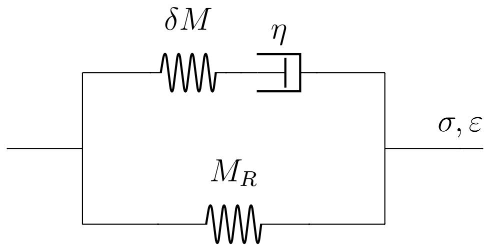


Figure 5.5: H-p-M Zener body.

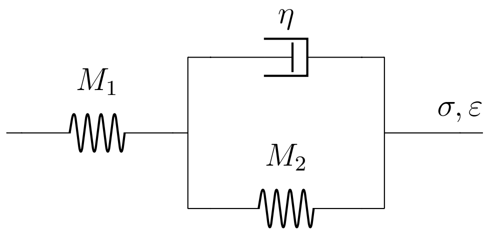


Figure 5.6: H-s-KV Zener body.

Zener body / standard linear body There are two really simple rheological models for viscoelastic behavior. The first is the Maxwell body, which consists of a series connection of the Hookes model and the Stokes model. Another simple possibility is the Kelvin-Voigt model, where the Hookes model and the Stokes model are connected in parallel. More general than Maxwell and Kelvin-Voigt bodies are the nevertheless relatively simple viscoelastic Zener bodies (also known as standard linear solid body). This model is more realistic for the representation of material media, such as rocks, polymers, and metals. There are two equivalent models that describe Zener models: The first one is the H-p-M model (Hooke body connected in parallel with Maxwell body) and and the second is the H-s-KV model (Hooke body connected in series with Kelvin-Voigt body), which are shown in figure 5.5 and 5.6. We discuss the H-p-M model, since it is easier to see the meaning of the elastic moduli in this one. At the time of the application of the unit-step strain the instantaneous, i.e., unrelaxed, stress will be given by the sum of moduli of the two elastic springs, $M_U = M_R + \delta M$. At the same time deformation of the dashpot will start to grow from zero. The growth of the viscous deformation will gradually release stress of the spring connected in series with the dashpot (i.e., spring in Maxwell body). In the limit, the relaxed stress, M_R, will be only in the spring connected in parallel with Maxwell body. We can easily derive the basic characteristics of this rheological model.

First, we need the physical relationships for serial and parallel circuits with two elements with the properties ε_1, σ_1 and ε_2, σ_2. For parallel connected components σ and ε for the full model are given by

$$\sigma = \sigma_1 + \sigma_2, \tag{5.24}$$

$$\varepsilon = \varepsilon_1 = \varepsilon_2. \tag{5.25}$$

Furthermore, for serial connected components σ and ε for the full model can be determined by

$$\sigma = \sigma_1 = \sigma_2, \tag{5.26}$$

$$\varepsilon = \varepsilon_1 + \varepsilon_2. \tag{5.27}$$

We proceed one by one to calculate the total strain and stress and calculate single strain and stress for the upper (σ_{top} and ε_{top}) and lower part (σ_{down} and $\varepsilon_{\text{down}}$) of the parallel circuit. With the equations 5.24 and 5.25 we get all in all

$$\sigma = \sigma_{\text{top}} + \sigma_{\text{down}}, \tag{5.28}$$

$$\varepsilon = \varepsilon_{\text{top}} = \varepsilon_{\text{down}}. \tag{5.29}$$

The upper part consists of a series connection of the elastic spring δM (Hookes body) and the dashpot η (Stokes body). We denote the stress and strain for the spring with σ_{HB} and ε_{HB}, and for the dashpot with σ_{SB} and ε_{SB}. With the equations (5.26) and (5.27) we get

$$\sigma_{\text{top}} = \sigma_{HB} = \sigma_{SB}, \tag{5.30}$$

$$\varepsilon_{\text{top}} = \varepsilon_{HB} + \varepsilon_{SB}, \tag{5.31}$$

and further using equations (5.5) and (5.7) we obtain that

$$\sigma_{HB} = \delta M \varepsilon_{HB}, \tag{5.32}$$

$$\sigma_{SB} = i\omega\eta\varepsilon_{SB}. \tag{5.33}$$

Now we use equation (5.30) and rewrite equations (5.32) and (5.34) as

$$\varepsilon_{HB} = \frac{\sigma_{\text{top}}(\omega)}{\delta M}, \tag{5.34}$$

$$\varepsilon_{SB} = \frac{\sigma_{\text{top}}(\omega)}{i\omega\eta}. \tag{5.35}$$

Inserting equations (5.34) and (5.35) into (5.31) yields

$$\varepsilon_{\text{top}}(\omega) = \frac{\sigma_{\text{top}}(\omega)}{\delta M} + \frac{\sigma_{\text{top}}(\omega)}{i\omega\eta} \tag{5.36}$$

and

$$\sigma_{\text{top}}(\omega) = \frac{i\omega\eta\delta M}{\delta M + i\omega\eta}\varepsilon_{\text{top}}(\omega). \tag{5.37}$$

Since the lower part only consists of one Hooke body, equation (5.5) leads to

$$\sigma_{\text{down}}(\omega) = M_R\varepsilon_{\text{top}}(\omega). \tag{5.38}$$

All in all, we insert equations (5.37) and (5.38) into equation (5.28) and use the identity in equation (5.29) and get

$$\sigma(\omega) = M_R\varepsilon(\omega) + \frac{i\omega\eta\delta M}{\delta M + i\omega\eta}\varepsilon(\omega). \tag{5.39}$$

We extend the fraction with $\frac{1}{\delta M}$ and bring it down to a common denominator and receive

$$\sigma(\omega) = \frac{M_R(1 + i\omega\frac{\eta}{\delta M}) + i\omega\eta}{1 + i\omega\frac{\eta}{\delta M}}. \tag{5.40}$$

Then we use the identity $\frac{M_R}{\delta M} = \frac{M_U}{\delta M} - 1$ and get

$$\sigma(\omega) = M_R\frac{1 + i\omega\frac{\eta}{\delta M}\frac{M_U}{M_R}}{1 + i\omega\frac{\eta}{\delta M}}\varepsilon(\omega). \tag{5.41}$$

We can see the stress and strain relaxation times from equation (5.41) and define τ_σ and τ_ε as

$$\tau_\sigma = \frac{\eta}{\delta M} \quad \text{and} \quad \tau_\varepsilon = \frac{\eta}{\delta M}\frac{M_U}{M_R}. \tag{5.42}$$

Using these definitions, we obtain the same stress strain relation as we have expected (see equation (5.13)) namely

$$\sigma(\omega) = M(\omega)\varepsilon(\omega) \text{ with } M(\omega) = M_R\frac{1 + i\omega\tau_\varepsilon}{1 + i\omega\tau_\sigma}. \tag{5.43}$$

By taking limits of $M(\omega)$, we verify our interpretation of the meaning of the elastic moduli:

$$\begin{aligned}\lim_{\omega\to\infty} M(\omega) &= M_R\frac{\tau_\varepsilon}{\tau_\sigma} = M_U = M_R + \delta M\\ \lim_{\omega\to 0} M(\omega) &= M_R\end{aligned}$$

Using equations (5.42) yields the simple relation between the unrelaxed and relaxed moduli:

$$M_U = M_R\frac{\tau_\varepsilon}{\tau_\sigma}$$

Finally, we can determine the stress relaxation function using equations (5.16) and (5.43):

$$\Psi(t) = \mathcal{F}^{-1}\left\{\frac{M(\omega)}{i\omega}\right\} = \mathcal{F}^{-1}\left\{M_R\left[\frac{-i}{\omega} + \frac{i\tau_\varepsilon}{i - \tau_\sigma\omega} - \frac{i\tau_\sigma}{i - \tau_\sigma\omega}\right]\right\} \tag{5.44}$$

It is now easy to find (details can be found in the appendix A.1.1) that

$$\Psi(t) = M_R\left[1 - \left(1 - \frac{\tau_\varepsilon}{\tau_\sigma}\right)\exp(-t/\tau_\sigma)\right]H(t),$$

where $H(t)$ is the Heavyside step function. More details on how to get to the decomposition in equation (5.44) and the execution of the Fourier transform are described in the appendix (see section A.1.1). Thus we have derived a rheological model and a relaxation

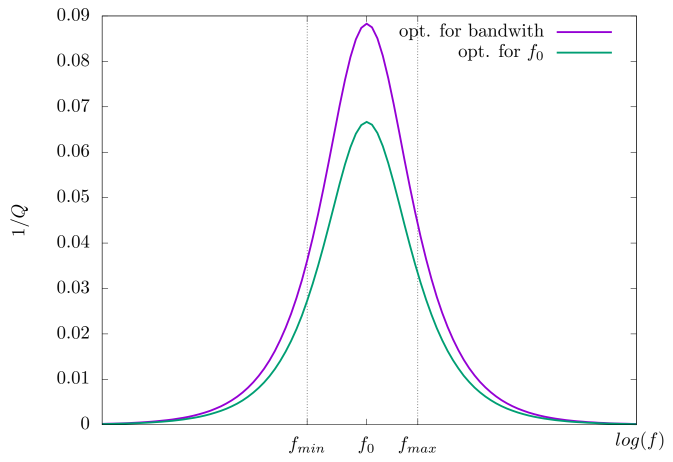


Figure 5.7: Inverse Q factor for the Zener model. Green line: relaxation parameters computed for f_0; purple line: relaxation parameters computed for frequency range $[f_{\min}, f_{max}]$.

function for viscoelastic media. We could use and insert this relaxation function into equation (5.3), and this would give us the forward problem. However, this model can only be used for a single frequency or a very small frequency band for which the relaxation parameters τ_ε and τ_σ have been determined. In this thesis, we consider Q factors which are constant in a given frequency band. Figure 5.7 shows the dissipation factor (the inverse Q factor). Here the green curve corresponds to the inverse Q factor if the model is determined for the frequency f_0 only and the purple curve for the frequency band of $[f_{\min}, f_{max}]$. We can see that with this model the dissipation factor can only be defined on a very small frequency band. On a wider frequency band, the deviation from the desired Q factor would be very large (the desired Q factor in this example is 15), because the approximation of the dissipation before and after the peak increases or decreases exponentially.

Generalized Zener-body As described at the end of the last section, the Zener model approach can only have a very small bandwidth. Some processes, such as grain boundary relaxation, have a dissipation factor that is much wider than a single relaxation curve. This can be achieved by connecting several Zener models in parallel as shown in figure 5.8. From the two equivalent models of the Zener body (see figure 5.5 and 5.6) we choose the one in which a single Zener body is of the H-p-M type (Hooke element in parallel with Maxwell body). In this model, we can immediately see the meaning ($M_{R,l}$, δM_l) of the elastic moduli of both Hooke elements in each Zener body. For the generalized Zener

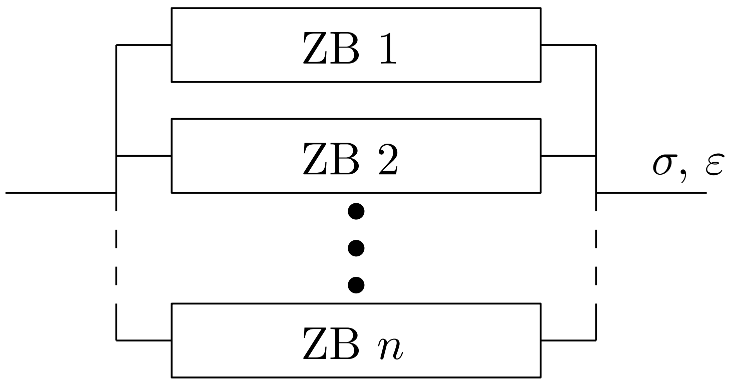


Figure 5.8: Generalized Zener body.

body, we easily obtain the modulus function

$$M(\omega) = \sum_{l=1}^{N_{\mathrm{SLS}}} M_{R,l} \frac{1 + i\tau_{\varepsilon,l}\omega}{1 + i\tau_{\sigma,l}\omega},$$

where N_{SLS} is the number of parallel Zener bodies, with relaxation times

$$\tau_{\varepsilon,l} = \frac{\eta_l}{\delta M_l}\frac{M_{U,l}}{M_{R,l}}\ ,\ \tau_{\sigma,l} = \frac{\eta_l}{\delta M_l} \quad \text{and} \quad \frac{\tau_{\varepsilon,l}}{\tau_{\sigma,l}} = \frac{M_{U,l}}{M_{R,l}}.$$

Furthermore, we know the relation between relaxed and unrelaxed moduli, that is

$$M_{U,l} = M_{R,l} + \delta M_l.$$

Using equations (5.20) and (5.21) leads to the unrelaxed and relaxed moduli

$$M_R \equiv \lim_{\omega\to 0} M(\omega) = \sum_{l=1}^{N_{\mathrm{SLS}}} M_{R,l},$$
$$M_U \equiv \lim_{\omega\to\infty} M(\omega) = \sum_{l=1}^{N_{\mathrm{SLS}}} M_{R,l}\frac{\tau_{\varepsilon,l}}{\tau_{\sigma,l}} = M_R \sum_{l=1}^{N_{\mathrm{SLS}}} \delta M_l.$$

Furthermore, we use relation (5.16) and get the relaxation function

$$\Psi(t) = \left\{ \sum_{l=1}^{N_{\mathrm{SLS}}} M_{R,l} \left[1 - \left(1 - \frac{\tau_{\varepsilon,l}}{\tau_{\sigma,l}} \right) \exp(-t/\tau_{\sigma,l}) \right] \right\} H(t). \tag{5.45}$$

We use the assumption by Carcicone [40]

$$M_{R,l} = \frac{1}{N_{\mathrm{SLS}}} M_R, \tag{5.46}$$

that leads to a simplification but is not too restrictive. Using this simplification yields:

$$M(\omega) = \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1 + i\omega\tau_{\varepsilon,l}}{1 + i\omega\tau_{\sigma,l}} \tag{5.47}$$

$$\Psi(t) = M_R \left[1 - \frac{1}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{\varepsilon,l}}{\tau_{\sigma,l}}\right) \exp(-t/\tau_{\sigma,l})\right] H(t) \tag{5.48}$$

Finally, we can determine the unrelaxed and relaxed moduli. At zero frequency $\omega = 0$, we get the relaxed modulus as

$$M(0) = \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1 + 0 \cdot i\tau_{\varepsilon,l}}{1 + 0 \cdot i\tau_{\sigma,l}} = M_R \tag{5.49}$$

and at infinite frequency, $\omega = \infty$, we get the unrelaxed modules as

$$M(\infty) = \lim_{\bar{\omega}\to\infty} \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1 + i\bar{\omega}\tau_{\varepsilon,l}}{1 + i\bar{\omega}\tau_{\sigma,l}} = \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{\tau_{\varepsilon,l}}{\tau_{\varepsilon,l}} = M_U. \tag{5.50}$$

Let us consider, as at the end of the last section, the Q factor approximation, which is to assume the constant value 15 in a frequency band $[f_{\min}, f_{\max}]$. With the use of only one Zener model, this value could only be achieved for one frequency (see figure 5.7). The standard generalized Zener approach allows describing viscoelastic materials that have a Q factor over a broader frequency band. Figure 5.9 shows the inverse Q factor for a generalized Zener model with three parallel Zener models. The individual contributions of the three Zener models are also shown there. With this approach, an almost constant Q factor can be simulated on a frequency band $[f_{\min}, f_{max}]$. The more parallel Zener models are used, the more constant is the Q factor in the desired frequency range.

Hence, we have everything we need to describe the wave propagation in viscoelastic material. Before we do this in the next section, we want to describe a disadvantage of this approach.

Disadvantage of this approach The generalized Zener model is well suited to describe homogeneous media, i.e., media that has the same attenuation properties everywhere. Materials that are not homogeneous would have to be described with different generalized Zener models (at least with the constant Q factor approach). Another problem is that the Q factors, which are used to determine the relaxation parameters, do not appear in the forward equations, as we will see in the next chapter. Therefore a full waveform inversion where the Q factors are inversion parameters is not possible.

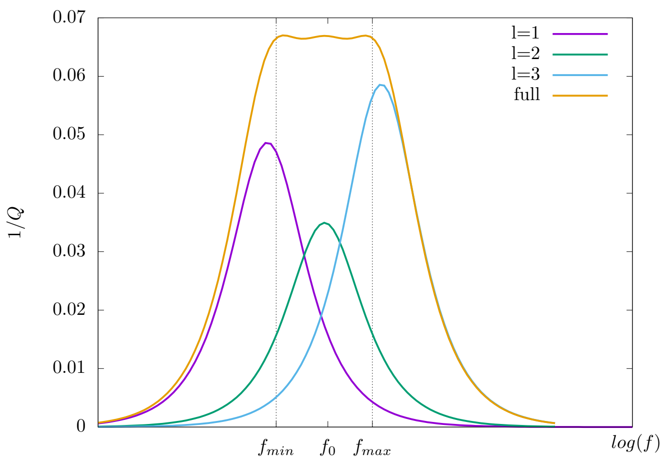


Figure 5.9: Inverse Q factor for the generalized Zener model with $N_{\text{SLS}} = 3$. Yellow line: resulting inverse Q factor; purple, green and blue: inverse Q factors for the three parallel Zener models.

5.2.3 Forward model

After we have derived a rheological model for viscoelastic material, we can use the relaxation function (5.48) in our mathematical model. In the following we put everything together, use partial integration to simplify equations and summarize everything to get the final equations. We start by applying equation (5.3) to equation (5.2) and get

$$\sigma_{ij}(x,t) = \int_{-\infty}^{t} \left\{ \frac{1}{2} \left[\Psi_1(x,t-t') - \Psi_2(x,t-t') \right] \delta_{ij} \partial_t \varepsilon_{kk}(t') + \Psi_2(x,t-t') \partial_t \varepsilon_{ij}(t') dt' \right\}.$$

Then we use equation (5.48) for the relaxation functions Ψ_1 and Ψ_2. Here, we mark the parameters belonging to the dilatational deformation relaxation function with a prefixed 1 in the subscript, so that M_R, $\tau_{\varepsilon,l}$, $\tau_{\sigma,l}$ is denoted by M_{1R}, $\tau_{1\varepsilon,l}$, $\tau_{1\sigma,l}$. The parameters for the relaxation function for the shear deformations are marked with 2. This leads to:

$$\begin{aligned}
\sigma_{ij}(x,t) = \int_{-\infty}^{t} \Bigg\{ & \frac{1}{2} M_{1R} \left(1 - \frac{1}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}} \right) \exp(-(t-t')/\tau_{1\sigma,l}) \right) \delta_{ij} \partial_t \varepsilon_{kk}(t') \\
& -\frac{1}{2} M_{2R} \left(1 - \frac{1}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}} \right) \exp(-(t-t')/\tau_{2\sigma,l}) \right) \delta_{ij} \partial_t \varepsilon_{kk}(t') \\
& +M_{2R} \left[1 - \frac{1}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}} \right) \exp(-(t-t')/\tau_{2\sigma,l}) \right] \partial_t \varepsilon_{ij}(t') \Bigg\} dt'.
\end{aligned}$$

The next step is to split the integrals into one part without exponantial functions and one integral with the exponantial functions

$$\begin{aligned}\sigma_{ij}(x,t) = &\int_{-\infty}^{t} \frac{1}{2} M_{1R} \delta_{ij} \partial_t \varepsilon_{kk}(t')dt' - \int_{-\infty}^{t} \frac{1}{2} \frac{M_{1R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \delta_{ij} \partial_t \varepsilon_{kk}(t')dt' \\ &- \int_{-\infty}^{t} \frac{1}{2} M_{2R} \delta_{ij} \partial_t \varepsilon_{kk}(t')dt' + \int_{-\infty}^{t} \frac{1}{2} \frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \delta_{ij} \partial_t \varepsilon_{kk}(t')dt' \\ &+ \int_{-\infty}^{t} M_{2R} \partial_t \varepsilon_{ij}(t')dt' - \int_{-\infty}^{t} \frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \partial_t \varepsilon_{ij}(t')dt'\end{aligned}$$

We now apply partial integration to the terms with exponential functions and obtain

$$\begin{aligned}\sigma_{ij}(x,t) = &\underbrace{\int_{-\infty}^{t} \frac{1}{2} M_{1R} \delta_{ij} \partial_t \varepsilon_{kk}(t')dt'}_{=\frac{1}{2} M_{1R} \delta_{ij} \varepsilon_{kk}(t)} - \underbrace{\left[\frac{1}{2} \frac{M_{1R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \delta_{ij} \varepsilon_{kk}(t')\right]_{-\infty}^{t}}_{\frac{1}{2} \frac{M_{1R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} (1-\tau_{1\varepsilon,l}/\tau_{1\sigma,l}) \delta_{ij} \varepsilon_{kk}(t)} \\ &+ \int_{-\infty}^{t} \frac{1}{2} \frac{M_{1R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \delta_{ij} \varepsilon_{kk}(t')dt' \\ &- \underbrace{\int_{-\infty}^{t} \frac{1}{2} M_{2R} \delta_{ij} \partial_t \varepsilon_{kk}(t)dt'}_{=\frac{1}{2} M_{2R} \delta_{ij} \varepsilon_{kk}(t)} + \underbrace{\left[\frac{1}{2} \frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \delta_{ij} \varepsilon_{kk}(t')\right]_{-\infty}^{t}}_{\frac{1}{2} \frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} (1-\tau_{2\varepsilon,l}/\tau_{2\sigma,l}) \delta_{ij} \varepsilon_{kk}(t)} \\ &- \int_{-\infty}^{t} \frac{1}{2} \frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \delta_{ij} \varepsilon_{kk}(t')dt' \\ &+ \underbrace{\int_{-\infty}^{t} M_{2R} \partial_t \varepsilon_{ij}(t')dt'}_{=M_{2R} \varepsilon_{ij}(t)} - \underbrace{\left[\frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-\frac{t-t'}{\tau_{2\sigma,l}}) \varepsilon_{ij}(t')\right]_{-\infty}^{t}}_{\frac{1}{2} \frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} (1-\tau_{2\varepsilon,l}/\tau_{2\sigma,l}) \varepsilon_{ij}(t)} \\ &+ \int_{-\infty}^{t} \frac{M_{2R}}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-(t-t')/\tau_{2\sigma,l}) \varepsilon_{ij}(t')dt'\end{aligned}$$

Calculating all terms where it is possible and using the identity $M_U = M_R(1-\frac{1}{N_{\text{SLS}}}\sum_{l=1}^{N_{\text{SLS}}}(1-\frac{\tau_\varepsilon}{\tau_\sigma}))$ yields

$$\begin{aligned}
\sigma_{ij}(x,t) = &\frac{1}{2}M_{1U}\delta_{ij}\varepsilon_{kk}(t)\\
&+\int_{-\infty}^{t}\frac{1}{2}\frac{M_{1R}}{N_{\text{SLS}}}\sum_{l=1}^{N_{\text{SLS}}}\frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\exp(-(t-t')/\tau_{1\sigma,l})\delta_{ij}\varepsilon_{kk}(t')dt'\\
&-\frac{1}{2}M_{2U}\delta_{ij}\varepsilon_{kk}(t)\\
&-\int_{-\infty}^{t}\frac{1}{2}\frac{M_{2R}}{N_{\text{SLS}}}\sum_{l=1}^{N_{\text{SLS}}}\frac{1}{\tau_{2\sigma,l}}\left(1-\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)\exp(-(t-t')/\tau_{2\sigma,l})\delta_{ij}\varepsilon_{kk}(t')dt'\\
&+M_{2U}\varepsilon_{ij}(t)\\
&+\int_{-\infty}^{t}\frac{M_{2R}}{N_{\text{SLS}}}\sum_{l=1}^{N_{\text{SLS}}}\frac{1}{\tau_{2\sigma,l}}\left(1-\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)\exp(-(t-t')/\tau_{2\sigma,l})\varepsilon_{ij}(t')dt'
\end{aligned}$$

To achieve easier readability we set $\varepsilon_{ij}^d = \varepsilon_{ij} - \frac{1}{2}\delta_{ij}\varepsilon_{kk}(t)$ and define the so-called memory variables

$$\begin{aligned}
e_{1,l} &= \Phi_{1l} * \varepsilon_{kk}\\
e_{ij,l} &= \Phi_{2l} * \varepsilon_{ij}^d\\
\Phi_{\nu,l}(t) &= \frac{M_{\nu R}}{N_{\text{SLS}}}\frac{1}{\tau_{\nu\sigma,l}}\left(1-\frac{\tau_{\nu\varepsilon,l}}{\tau_{\nu\sigma,l}}\right)\exp(-t/\tau_{\nu\sigma,l}) \qquad (\nu = 1,2)
\end{aligned}$$

which allows to write the stress tensor as

$$\sigma_{ij} = \frac{1}{2}\delta_{ij}\left(M_{1U}\varepsilon_{kk} + \sum_{l=1}^{N_{\text{SLS}}} e_{1,l}\right) + \left(M_{2U}\varepsilon_{ij}^d + \sum_{l=1}^{N_{\text{SLS}}} e_{ij,l}\right).$$

At this point, we have everything together and only have to insert the moduli and can combine terms. The modulus for the dilatational deformations is $M_1 = 2\lambda + 2\mu$ and for the shear deformation is $M_2 = 2\mu$, where λ and μ are the Lamé parameters. To achieve better readability, we skip the subscript U to describe unrelaxed parameters, we only use the subscript R to indicate relaxed parameters. That means $\lambda = \lambda_U$ describes an unrelaxed parameter, and λ_R describes the relaxed equivalent. This results in the following components of the stress tensor:

$$\sigma_{xx} = \frac{1}{2}\left((2\lambda + 2\mu)\varepsilon_{kk} + \sum_{l=1}^{N_{\text{SLS}}} e_{1,l}\right) + \left(2\mu\varepsilon_{xx}^d + \sum_{l=1}^{N_{\text{SLS}}} e_{xx,l}\right)$$
$$\sigma_{yy} = \frac{1}{2}\left((2\lambda + 2\mu)\varepsilon_{kk} + \sum_{l=1}^{N_{\text{SLS}}} e_{1,l}\right) + \left(2\mu\varepsilon_{yy}^d + \sum_{l=1}^{N_{\text{SLS}}} e_{yy,l}\right)$$
$$\sigma_{xy} = \left(2\mu\varepsilon_{xy}^d + \sum_{l=1}^{N_{\text{SLS}}} e_{xy,l}\right)$$

Calculating and summarizing the bracket terms and using the definition of ε_{ij}^d results in:

$$\sigma_{xx} = \lambda\varepsilon_{kk} + \mu\varepsilon_{kk} + \frac{1}{2}\sum_{l=1}^{N_{\text{SLS}}} e_{1,l} + 2\mu(\varepsilon_{xx} - \frac{1}{2}\delta_{xx}\varepsilon_{kk}) + \sum_{l=1}^{N_{\text{SLS}}} e_{xx,l}$$
$$\sigma_{yy} = \lambda\varepsilon_{kk} + \mu\varepsilon_{kk} + \frac{1}{2}\sum_{l=1}^{N_{\text{SLS}}} e_{1,l} + 2\mu(\varepsilon_{yy} - \frac{1}{2}\delta_{yy}\varepsilon_{kk}) + \sum_{l=1}^{N_{\text{SLS}}} e_{yy,l}$$
$$\sigma_{xy} = 2\mu(\varepsilon_{xy} - \frac{1}{2}\delta_{xy}\varepsilon_{kk}) + \sum_{l=1}^{N_{\text{SLS}}} e_{xy,l}$$

Summarizing again finally leads to

$$\sigma_{xx} = (\lambda + 2\mu)\varepsilon_{xx} + \lambda\varepsilon_{yy} + \frac{1}{2}\sum_{l=1}^{N_{\text{SLS}}} e_{1,l} + \sum_{l=1}^{N_{\text{SLS}}} e_{xx,l} \tag{5.51}$$

$$\sigma_{yy} = \lambda\varepsilon_{xx} + (\lambda + 2\mu)\varepsilon_{yy} + \frac{1}{2}\sum_{l=1}^{N_{\text{SLS}}} e_{1,l} + \sum_{l=1}^{N_{\text{SLS}}} e_{yy,l} \tag{5.52}$$

$$\sigma_{xy} = 2\mu\varepsilon_{xy} + \sum_{l=1}^{N_{\text{SLS}}} e_{xy,l} \tag{5.53}$$

But before we write down the PDE system, we need a differential equation for the memory variables. Furthermore, we can also combine two of the memory variables. Therefore, we use equation (5.50), i.e., $M_{\nu R} = N_{\text{SLS}} \frac{M_{\nu U}}{\sum_{l=1}^{N_{\text{SLS}}} \frac{\tau_{\nu\varepsilon,l}}{\tau_{\nu\sigma,l}}}$, and apply $M_1 = 2\lambda + 2\mu$ and $M_2 = 2\mu$ to the memory variables, and see that we only need one memory variable for $e_{xx,l}$ and $e_{yy,l}$

instead of two, because $e_{xx,l} = -e_{yy,l}$. The memory variables can be computed by:

$$\begin{aligned}
e_{1,l} &= \Phi_{1l} * \varepsilon_{kk} \\
&= \int_{-\infty}^{t} \frac{M_{1R}}{N_{\text{SLS}}} \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \exp(-(t-t')/\tau_{1\sigma,l})(\partial_x u_x + \partial_y u_y) \\
&= \int_{-\infty}^{t} \frac{2\lambda + 2\mu}{\sum_{l=1}^{N_{\text{SLS}}} \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}} \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \exp(-(t-t')/\tau_{1\sigma,l})(\partial_x u_x + \partial_y u_y)
\end{aligned}$$

$$\begin{aligned}
e_{xx,l} &= \Phi_{2l} * \varepsilon_{xx}^{d} \\
&= \int_{-\infty}^{t} \frac{M_{2R}}{N_{\text{SLS}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-(t-t')/\tau_{2\sigma,l})(\varepsilon_{xx} - \frac{1}{2}\delta_{xx}\varepsilon_{kk}) \\
&= \int_{-\infty}^{t} \frac{2\mu}{\sum_{l=1}^{N_{\text{SLS}}} \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-(t-t')/\tau_{2\sigma,l})(\frac{1}{2}(\partial_x u_x - \partial_y u_y))
\end{aligned}$$

$$\begin{aligned}
e_{yy,l} &= \Phi_{2l} * \varepsilon_{yy}^{d} \\
&= \int_{-\infty}^{t} \frac{M_{2R}}{N_{\text{SLS}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-(t-t')/\tau_{2\sigma,l})(\varepsilon_{yy} - \frac{1}{2}\delta_{yy}\varepsilon_{kk}) \\
&= \int_{-\infty}^{t} \frac{2\mu}{\sum_{l=1}^{N_{\text{SLS}}} \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-(t-t')/\tau_{2\sigma,l})(-\frac{1}{2}(\partial_x u_x - \partial_y u_y)) \\
&= -e_{xxl}
\end{aligned}$$

$$\begin{aligned}
e_{xy,l} &= \Phi_{2l} * \varepsilon_{xy}^{d} \\
&= \int_{-\infty}^{t} \frac{M_{2R}}{N_{\text{SLS}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-(t-t')/\tau_{2\sigma,l})\varepsilon_{xy} \\
&= \int_{-\infty}^{t} \frac{2\mu}{\sum_{l=1}^{N_{\text{SLS}}} \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}} \frac{1}{\tau_{2\sigma,l}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \exp(-(t-t')/\tau_{2\sigma,l})\frac{1}{2}(\partial_y u_x + \partial_x u_y))
\end{aligned}$$

Since the moduli are independent of the memory variables, we put them in front of the integrals and define

$$
\begin{aligned}
R_{1,l} :=& \frac{1}{2}\frac{\sum_{l=1}^{N_{\text{SLS}}}\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}}{\lambda+\mu} e_{1,l} &\Leftrightarrow&& 2\frac{\lambda+\mu}{\sum_{l=1}^{N_{\text{SLS}}}\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}} R_{1,l} = e_{1,l} \\
R_{xx,l} :=& 2\frac{\sum_{l=1}^{N_{\text{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}{2\mu} e_{xx,l} &\Leftrightarrow&& \frac{\mu}{\sum_{l=1}^{N_{\text{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}} R_{xx,l} = e_{xx,l} \\
R_{xy,l} :=& 2\frac{\sum_{l=1}^{N_{\text{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}{2\mu} e_{xy,l} &\Leftrightarrow&& \frac{\mu}{\sum_{l=1}^{N_{\text{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}} R_{xy,l} = e_{xy,l}
\end{aligned}
$$

Now as the last step we need an update scheme for the memory variables, or in other words, we need PDEs for the memory variables. We derive a PDE exemplary for $R_{1,l}$:

$$
\begin{aligned}
R_{1,l} =& \frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \operatorname{tr}(\varepsilon)(\cdot)\right) \\
\partial_t R_{1,l} =& \frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\partial_t\left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \operatorname{tr}(\varepsilon)(\cdot)\right) \\
=& \frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \partial_t\operatorname{tr}(\varepsilon)(\cdot)\right) \\
=& \frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\left[\exp(-(t-t')/\tau_{1\sigma,l})\operatorname{tr}(\varepsilon)\right]_{-\infty}^{t} \\
&-\frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\int_{-\infty}^{t}\frac{1}{\tau_{1\sigma,l}}\exp(-(t-t')/\tau_{1\sigma,l})\operatorname{tr}(\varepsilon)dt' \\
=& \frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\left[\exp(0)\operatorname{tr}(\varepsilon)-0\right] \\
&-\frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\underbrace{\int_{-\infty}^{t}\frac{1}{\tau_{1\sigma,l}}\exp(-(t-t')/\tau_{1\sigma,l})\operatorname{tr}(\varepsilon)dt'}_{=\frac{1}{\tau_{1\sigma,l}}\left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}})*\operatorname{tr}(\varepsilon)(\cdot)\right)} \\
=& -R_{1,l}/\tau_{1\sigma,l}+\operatorname{tr}(\varepsilon)\frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)
\end{aligned}
$$

From the first to the second line, we differentiate the equation with respect to t. In the third line, we used the rule for convolutions that $\partial_t(f*g) = (\partial_t f)*g = f*\partial_t g$. Then we use integration by parts resulting in line four. Inserting the integration bounds and substituting $R_{1,l}$ results in the last line.

Thus we now have everything together to describe the wave propagation in viscoelastic material. Using equation (5.1) and equations (5.51)–(5.53), this results in the following second-order PDE system:

$$
\begin{aligned}
\rho\partial_{tt}u_i &= u_i\partial_j\sigma_{ij} + f_i \\
\sigma_{xx} &= (\lambda+2\mu)\partial_x u_x + \lambda\partial_y u_y + \frac{\lambda+\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} + \frac{\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} \\
\sigma_{yy} &= \lambda\partial_x u_x + (\lambda+2\mu)\partial_y u_y + \frac{\lambda+\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} - \frac{\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} \\
\sigma_{xy} &= \mu(\partial_x u_y + \partial_y u_x) + \frac{\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xy,l}
\end{aligned}
$$

We can rewrite this easily to a system of first order equations (see section A.1.2 in the appendix for more details) and get:

$$\rho\partial_t v_x = \partial_x\sigma_{xx} + \partial_y\sigma_{xy} + f_x \tag{5.54}$$

$$\rho\partial_t v_y = \partial_x\sigma_{xy} + \partial_y\sigma_{yy} + f_y \tag{5.55}$$

$$\partial_t\sigma_{xx} = (\lambda+2\mu)\partial_x v_x + \lambda\partial_y v_y + \frac{\lambda+\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} + \frac{\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} \tag{5.56}$$

$$\partial_t\sigma_{yy} = \lambda\partial_x v_x + (\lambda+2\mu)\partial_y v_y + \frac{\lambda+\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} - \frac{\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} \tag{5.57}$$

$$\partial_t\sigma_{xy} = \mu(\partial_x v_y + \partial_y v_y) + \frac{\mu}{\sum_{l=1}^{N_{\mathrm{SLS}}}\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xy,l} \tag{5.58}$$

$$\partial_t R_{1,l} = -R_{1,l}/\tau_{1\sigma,l} + \frac{1}{\tau_{1\sigma,l}}\left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)(\partial_x v_x + \partial_y v_y) \tag{5.59}$$

$$\partial_t R_{xx,l} = -R_{xx,l}/\tau_{2\sigma,l} + \frac{1}{\tau_{2\sigma,l}}\left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)(\partial_x v_x - \partial_y v_y) \tag{5.60}$$

$$\partial_t R_{xy,l} = -R_{xy,l}/\tau_{2\sigma,l} + \frac{1}{\tau_{2\sigma,l}}\left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)(\partial_x v_y + \partial_y v_x) \tag{5.61}$$

Note that the auxiliary variables (e.g., $R_{1,l}$) are not the same as for the second-order PDE, but for simplicity's sake, we name them the same. We recall that equations (5.59)–(5.61) are each N_{SLS} equations.

5.2.4 Relation between Zener model and physical parameters

As we have seen before, viscoelastic numerical models, based on a concept of dissipation mechanisms, take into account memory variables, and some dissipation parameters, namely stress and strain relaxation times. In geophysical practice, the quality factor Q is widely used for describing an attenuation property of viscoelastic media. That means that we need a relation between the dissipation parameters and the quality factor, which we can use to compute the dissipation parameters.

We recall the definition of Q (cf. equation (5.23))

$$Q(\omega) = \frac{\Re(M(\omega))}{\Im(M(\omega))}.$$

The real and the imaginary part of the moduli function are

$$\Re(M(\omega)) = \frac{M_R}{N_{\mathrm{SLS}}} \left(\sum_{l=1}^{N_{\mathrm{SLS}}} \frac{1 + \omega^2 \tau_{\sigma,l} \tau_{\varepsilon,l}}{1 + \omega^2 {\tau_{\sigma,l}}^2} \right),$$

respectively

$$\Im(M(\omega)) = \frac{M_R}{N_{\mathrm{SLS}}} \left(\sum_{l=1}^{N_{\mathrm{SLS}}} \frac{\omega(\tau_{\varepsilon,l} - \tau_{\sigma,l})}{1 + \omega^2 {\tau_{\sigma,l}}^2} \right).$$

We insert both into equation (5.23) and get

$$Q(\omega) = \left(\sum_{l=1}^{N_{\mathrm{SLS}}} \frac{1 + \omega^2 \tau_{\sigma,l} \tau_{\varepsilon,l}}{1 + \omega^2 {\tau_{\sigma,l}}^2} \right) \cdot \left(\sum_{l=1}^{N_{\mathrm{SLS}}} \frac{\omega(\tau_{\varepsilon,l} - \tau_{\sigma,l})}{1 + \omega^2 {\tau_{\sigma,l}}^2} \right)^{-1} \tag{5.62}$$

Next, we transform equation (5.62) into a minimization problem to determine the dissipation parameters.

Minimization problem to obtain relaxation parameters To obtain the stress and strain relaxation times for a given Q factor, we need to solve an additional minimization problem (see e.g., [26]). Therefore, we define

$$\theta_l := \frac{1}{\tau_{\sigma,l}} \quad \text{and} \quad d_l := \frac{1}{N_{\mathrm{SLS}}} \left(\frac{\tau_{\varepsilon,l}}{\tau_{\sigma,l}} - 1 \right) \tag{5.63}$$

and the back substitution is

$$\tau_{\varepsilon,l} = \frac{1 + N_{\mathrm{SLS}} d_l}{\theta_l} \quad \text{and} \quad \tau_{\sigma,l} = \frac{1}{\theta_l}. \tag{5.64}$$

Using this variables the reciprocal Q factor can be written by

$$Q^{-1} = \frac{\sum_{l=1}^{N_{\mathrm{SLS}}} \frac{d_l \omega \theta_l}{\theta_l^2 + \omega^2}}{\left(1 + \sum_{l=1}^{N_{\mathrm{SLS}}} \frac{d_l \omega^2}{\theta_l^2 + \omega^2} \right)}.$$

The first step to get an optimization problem is to resolve the fraction and the bracket:

$$Q^{-1}\left(1+\sum_{l=1}^{N_{\text{SLS}}}\frac{d_l\omega^2}{\theta_l^2+\omega^2}\right)=\sum_{l=1}^{N_{\text{SLS}}}\frac{d_l\omega\theta_l}{\theta_l^2+\omega^2}$$

$$Q^{-1}=\sum_{l=1}^{N_{\text{SLS}}}\frac{d_l\omega\theta_l-Q^{-1}\omega^2 d_l}{\omega^2+\omega_l^2}.$$

Now we subtract the Q factor we are looking for, called Q_{ref}, integrate over the desired frequency band and obtain our minimization problem:

$$\begin{aligned}\chi &= \int (Q^{-1}(x,\omega)-Q_{\text{ref}}^{-1}(x))^2\mathrm{d}\omega\\ &= \int\left(\sum_{l=1}^{N_{\text{SLS}}}\frac{d_l\omega\theta_l-Q_{\text{ref}}^{-1}\omega^2 d_l}{\theta_l^2+\omega^2}-Q_{\text{ref}}^{-1}(x)\right)^2\mathrm{d}\omega\\ &= \int\left(\sum_{l=1}^{N_{\text{SLS}}}\frac{Q_{\text{ref}}(x)d_l\omega\theta_l-\omega^2 d_l}{\theta_l^2+\omega^2}-1\right)^2\mathrm{d}\omega\end{aligned}$$

Remember that we consider constant Q factors over frequency, so Q_{ref} is independent of frequency. To obtain the final minimization problem, we discretize the integral and obtain

$$\mathcal{J}(\{d_l,\theta_l\};N_{\text{SLS}},K)=\sum_{k=1}^{K}\left(\sum_{l=1}^{N_{\text{SLS}}}\frac{\omega_k Q_{\text{ref}}(\omega_k)(\theta_l-\omega_k Q_{\text{ref}}^{-1}(\omega_k))}{\theta_l^2+\omega_k^2}d_l-1\right)^2 \tag{5.65}$$

where ω_k are the discretized frequencies, K is the total number them.

Disadvantage of this approach The approach described in the last section has two disadvantages. The first results from the fact that stress and strain relaxation times are domain dependent. Therefore, in the worst case, the solution of a minimization problem is necessary for each discrete domain point. The second disadvantage is that the quality factors do not appear directly in the equations. Then it is not possible to use Q as an optimization parameter for waveform inversion (see section 5.4), since it is not possible to determine the partial derivatives for Q_1 or Q_2. There are different approaches to overcome these disadvantages [52, 53, 54, 158], we follow the one by van Driel and Fichtner [54].

5.3 Waveform modeling

The standard approach for the forward modeling of wave propagation in viscoelastic material has already been described in the previous section 5.2.3. However, this approach has two disadvantages which we have described at the end of the last section. In this section, we present an approach that eliminates these disadvantages and which is the basis of the simulations in this thesis.

5.3.1 Modification of the standard approach

On the one hand, it should be ensured that the Q factors appear directly in the forward model, and not only indirectly via a minimization problem to be solved beforehand. And secondly, it should be ensured that the resulting relaxation parameters are valid not only for a specific Q factor, but for a range. The second point is of course a contradiction to the Zener model since the Zener model is valid for a special constant Q factor. This means that the relaxation parameters τ_ε and τ_σ or θ and d depend on Q or were determined for a specific Q. This becomes particularly clear when we look at the definition (5.42) of τ_ε and τ_σ. Both depend on $\delta M = M_U - M_R$, which is just a quantity described by the Q factor. However, we see below that for τ_σ the dependency is very small if the frequency band for which the Q factor is applied remains the same, so τ_σ can be neglected in this case. For this, we will investigate the influence of d and θ on the moduli function and the Q factor. First, we consider the unrelaxed moduli and apply the substitutions (5.63) to the definition (5.22) so that the unrelaxed modulus is given as

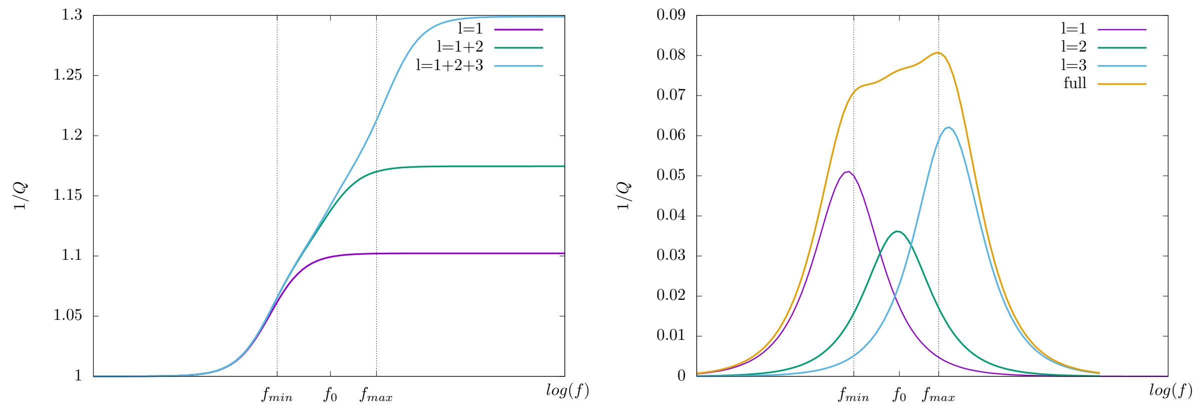
$$M_U = \lim_{\omega\to\infty} M_R \left(1 + \sum_{l=1}^{N_{\mathrm{SLS}}} \frac{i\omega d_l}{\theta_l^2 + i\omega}\right) = M_R \left(1 + \sum_{l=1}^{N_{\mathrm{SLS}}} d_l\right).$$

This shows that d_l must depend on the Q factor in any case. Next we consider the real and imaginary parts of the modulus function $M(\omega)$

$$\Re M(\omega) = M_R \left(1 + \sum_{l=1}^{N_{\mathrm{SLS}}} \frac{d_l\omega^2}{\theta_l^2 + \omega^2}\right)$$
$$\Im M(\omega) = M_R \left(\sum_{l=1}^{N_{\mathrm{SLS}}} \frac{d_l\theta_l\omega}{\theta_l^2 + \omega^2}\right)$$

and the Q factor, which are shown in figures 5.10a, 5.10b, 5.11 for $N_{\mathrm{SLS}} = 3$.

We can see that d_l only appears as a ‘scaling’ parameter. With this parameter, we can control how large the Q factor is. The parameter θ on the other hand shifts the functions in the frequency range. This is easiest to see for the imaginary part. In logarithmic



(a) Real part (b) Imaginary part

Figure 5.10: Real and imaginary part of modulus function for the generalized Zener model and the contribution of the three Zener bodies.

scaling, the real part can be displayed as a logistic function:

$$\Re M(\omega) = M_R\left(1 + \frac{d\omega^2}{\theta^2 + \omega^2}\right) = M_R + \frac{dM_R}{1 + \frac{\theta^2}{\omega^2}}$$

Substituting $\bar{\theta} := \log_{10}(\theta)$ and $\bar{\omega} = \log_{10}(\omega)$ yield

$$= M_R + \frac{dM_R}{1 + 10^{\bar{\theta}}10^{-\bar{\omega}^2}} = M_R + \frac{dM_R}{1 + 10^{-\left(\bar{\omega}^2 - \bar{\theta}\right)}},$$

which is exactly the definition of the logistic function, with function maximum value dM_R and center $\bar{\theta}$. Since we assume that the frequency band on which the Q factor affects is known and constant, and only the Q factor is unknown, we can consider θ to be independent of Q.

The plan is now to divide d_l into a part that is independent of Q and a part that depends on Q, so that Q appears directly in the equations of the forward model.

5.3.2 Minimization problem to obtain relaxation parameters

In the last section, we have seen that with the frequency range remaining the same, the Q factor in the Zener model can be scaled by the relaxation parameter d_l. Therefore we decompose d_l multiplicatively to

$$d_l := Q^{-1}D_l \tag{5.66}$$

with a Q independent parameter D_l.

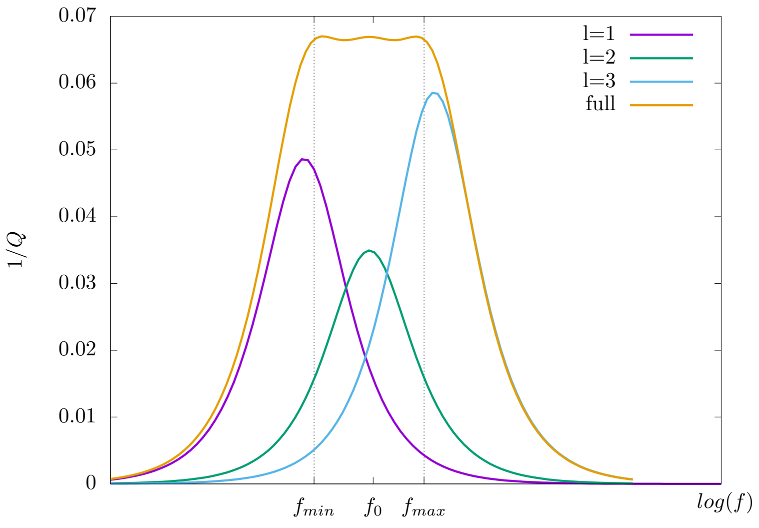


Figure 5.11: Inverse Q factor the generalized Zener model and the contribution of the three Zener bodies.

If we replace d_l with this decomposition, the minimization problem (5.65) becomes:

$$\begin{aligned}\bar{\mathcal{J}}(D_l,\theta_l) &= \sum_{k=1}^{K}\left(\sum_{l=1}^{N_{\text{SLS}}}\frac{\omega_k Q_{\text{ref}}(\omega_k)(\theta_l-\omega_k Q_{\text{ref}}^{-1}(\omega_k))}{\theta_l^2+\omega_k^2}d_l-1\right)^2\\ &= \sum_{k=1}^{K}\left(\sum_{l=1}^{N_{\text{SLS}}}\frac{\omega_k Q_{\text{ref}}(\omega_k)(\theta_l-\omega_k Q_{\text{ref}}^{-1}(\omega_k))}{\theta_l^2+\omega_k^2}Q_{\text{ref}}^{-1}D_l-1\right)^2\\ &= \sum_{k=1}^{K}\left(\sum_{l=1}^{N_{\text{SLS}}}D_l\frac{\theta_l\omega_k}{\omega_k^2+\theta_l^2}-Q_{\text{ref}}^{-1}D_l\frac{\omega_k^2}{\omega_k^2+\theta_l^2}-1\right)^2\end{aligned}$$

However, since the parameters D_l and θ_l are now independent of Q, this minimization problem can be extended so that it not only looks for the optimal parameters D_l and θ_l for Q_{ref}. It is looking for the optimal relaxation parameters D_l and θ_l which describes the underlying generalized Zener model for all $Q \in [Q_{\min}, Q_{max}]$. For this purpose, we divide the Q factor range into N equidistant values $Q_{\min} = Q_1, Q_2, \ldots, Q_N = Q_{max}$ and get the resulting minimization problem

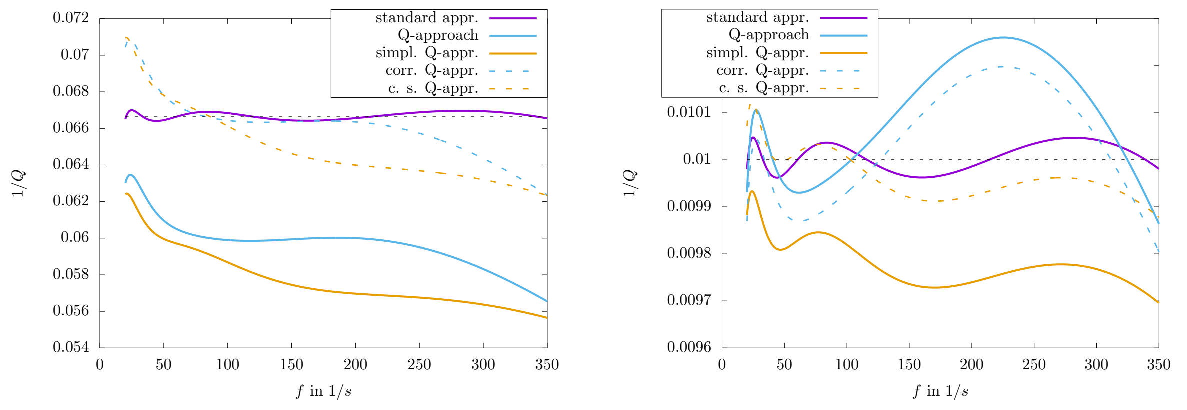
$$\mathcal{J}(D_l,\theta_l) = \sum_{n=1}^{N}\sum_{k=1}^{K}\left(\sum_{l=1}^{N_{\text{SLS}}}D_l\frac{\theta_l\omega_k}{\omega_k^2+\theta_l^2}-Q_n^{-1}D_l\frac{\omega^2}{\omega_k^2+\theta_l^2}-1\right)^2. \tag{5.67}$$

This minimization problem can be further simplified. The second term can be neglected for large Q factors or a very large Q range. This results in a simple minimization problem

$$\mathcal{J}_{simp.}(D_l,\theta_l) = \sum_{n=1}^{N}\sum_{k=1}^{K}\left(\sum_{l=1}^{N_{\text{SLS}}}D_l\frac{\theta_l\omega_k}{\omega_k^2+\theta_l^2}-1\right)^2. \tag{5.68}$$

5.3.3 Quality comparison and improvement of the Q factor approximation

Before we implement the modified Zener model into the forward model, we first test the quality of the model to see if it is suitable for simulating wave propagation in viscoelastic media. For this purpose we consider a frequency band $[20, 350]$ and want to simulate a rheological model for all Q factors $Q \in [15, 100]$, use generalized Zener bodies with $N_{\text{SLS}} = 3$ and set the number of discrete frequencies $K = 12$. Therefore, we use the SolvOpt [89] optimization algorithm (cf. section 5.5.4) to determine the relaxation parameters for the modified Zener model both for the complete objective function (5.67) and for the simplified objective function (5.68). First, we compare the approximation quality for the Q factor $Q_{\min} = 15$ with the approximation provided by the generalized Zener model. The approximations are shown in figure 5.12a, where the purple line is the approximation for the generalized Zener model, the blue line is the modified Zener model when using the full objective function, the yellow line is the modified Zener model when using the simplified objective function, and the black dashed line represents the desired constant Q factor. Both Q factor approximations of the modified Zener model do not match any frequency in



(a) $Q_{\min}$ (b) $Q_{\max}$

Figure 5.12: Inverse Q factor for generalized Zener model and modified Zener model. The purple line is the approximation for the generalized Zener model, the blue line is the modified Zener model when using the full objective function, the yellow line is the modified Zener model when using the simplified objective function, and the black dashed line represents the desired constant Q factor. The blue and yellow dashed lines are the approximations using the piecewise linear correction for the full and the simplified objective function.

the frequency band with the Q factor to be approximated, but the approximation is better for the full objective function. A similar picture emerges if we compare the approximation for the Q factor $Q_{max} = 100$ with the approximation provided by the generalized Zener model. However, in this case, the modified Zener model with the complete objective function also approximates well and oscillates around the Q factor to be approximated and approximates it exactly at four frequencies.

Especially due to the observation for $Q_{\min}$ that the approximations for the modified

Zener model do not exactly approximate the desired Q factor at any point, we want to improve the model.

We recall that $Q^{-1}D_l$ is a kind of scaling parameter for the dissipation of the Zener model. Therefore there must be a parameter $Q_{\text{opt}} = Q_{\text{ref}} + \text{shift}$ that yields the best approximation for the desired Q_{ref}. Thus, we search for all $Q \in \{Q_{\min} = Q_1, Q_2, \ldots, Q_N = Q_{max}\}$ for the optimal Q_{opt}, so that

$$Q_{\text{ref}}^{-1} \overset{!}{\approx} \frac{1}{Q_{\text{opt}}} \left(\sum_{l=1}^{N_{\text{SLS}}} \frac{D_l \omega \theta_l}{\theta_l^2 + \omega^2} \right) \left(1 + \frac{1}{Q_{\text{opt}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{D_l \omega^2}{\theta_l^2 + \omega^2} \right)^{-1}$$

has the smallest maximum error. For our example, this is shown in figure 5.13. Here the green line is the shift for the modified model determined with the simplified objective function and the purple curve is the shift for the modified model with the full objective function. For the simplified objective function the shift is relatively constant over the whole Q range (around -1.9). For the full objective function, the shift ranges from -1.6 to 0.6 and can be divided into two affine linear parts.

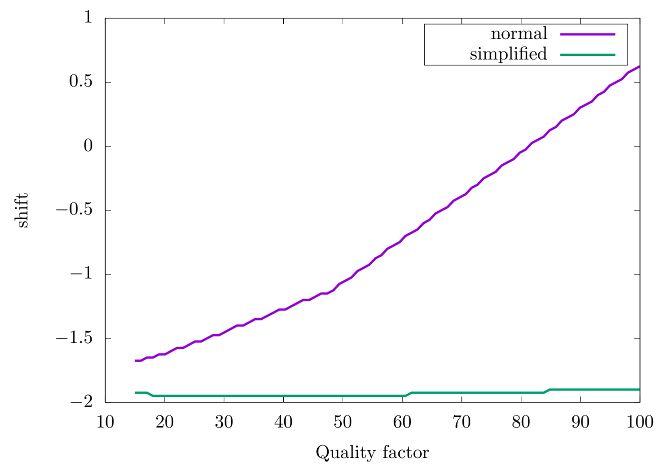
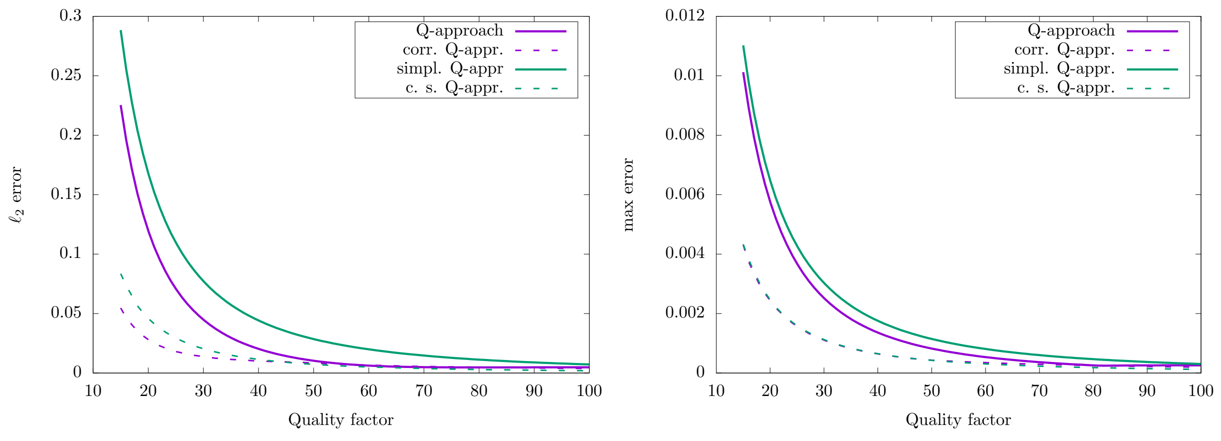


Figure 5.13: Shift to achieve the best Q factor approximation for the full (normal) and simplified objective function.

This numerical study shows that it is a good and efficient idea to replace Q_{ref} by a piecewise linear function $m \cdot Q_{\text{ref}} + b$:

$$Q_{\text{ref}}^{-1} \overset{!}{\approx} \frac{1}{m \cdot Q_{\text{ref}} + b} \left(\sum_{l=1}^{N_{\text{SLS}}} \frac{D_l \omega \theta_l}{\theta_l^2 + \omega^2} \right) \left(1 + \frac{1}{m \cdot Q_{\text{ref}} + b} \sum_{l=1}^{N_{\text{SLS}}} \frac{D_l \omega^2}{\theta_l^2 + \omega^2} \right)^{-1}$$

This piecewise linear correction leads to much smaller errors, especially for small Q factors. Figure 5.14a shows the square error and figure 5.14b shows the maximum error. For small Q factors, the model resulting from the full objective function remains better. However, if we look again at the inverse Q factors Q_{max} (figure 5.12b dashed lines) we can see that the modified model resulting from the simplified objective function, after improvement with the shift, approximates the Q factor better than the model resulting from the full



(a) ℓ_2 error (b) maximum error

Figure 5.14: ℓ_2 and maximum error for the standard objective function (purple) and the simplified objective function (green). The dashed lines are the errors using the piecewise correction.

objective function after correction.

The same behavior can be observed on larger Q intervals ($[15, 500]$, $[15, 900]$, $[800, 900]$ and $[500, 1000]$) and more parallel Zener models ($N_{\text{SLS}} = 10$), as shown in the figures in the appendix A.1.5. For larger frequency ranges there are more than two linear regions. Therefore, a division of the frequency range and different dissipation parameters for the different ranges can lead to a better approximation of the Q factor.

5.3.4 Modified forward model

After we have eliminated the two disadvantages of the standard approach and improved our modified model, we can now give the final forward equations. For the sake of clarity, we will only write Q_1 and Q_2, but we always mean the improvement with the piecewise linear offset presented in the previous section. Therefore, we approximate Q with two affine parts.

We obtain the relaxation functions for the modified Zener model by inserting the substitutions (5.66) and (5.63) into the equation (5.48) and get

$$\Psi(t) = M_R \left(1 + Q^{-1} \sum_{l=1}^{N_{\text{SLS}}} D_l \exp(-t\theta_l)\right) H(t). \tag{5.69}$$

If we use the same substitutions in the modulus function (5.47) we get

$$M(\omega) = M_R \left(1 + Q^{-1} \sum_{l=1}^{N_{\text{SLS}}} \frac{i\omega D_l(\theta_l - i\omega)}{\theta_l^2 + \omega^2}\right). \tag{5.70}$$

Furthermore, the unrelaxed moduli are still needed, which result with definition (5.22) as follows

$$M_U = \lim_{\bar{\omega}\to\infty} M_R \left(1 + Q^{-1} \sum_{l=1}^{N_{\mathrm{SLS}}} \frac{i\bar{\omega}\theta_l D_l - i\bar{\omega}^2 D_l}{\theta_l^2 + \bar{\omega}^2}\right) = M_R \left(1 + Q^{-1} \sum_{l=1}^{N_{\mathrm{SLS}}} D_l\right). \tag{5.71}$$

The detailed translations can be found in the appendix A.1.3.

For this new approach we get the forward equations in the same way as we have derived equations (5.54)–(5.61). Thus equations (5.2), (5.3) and (5.69) result in the following system of equations:

$$\rho\partial_t v_x = \partial_x\sigma_{xx} + \partial_y\sigma_{xy} + f_x \tag{5.72}$$

$$\rho\partial_t v_y = \partial_x\sigma_{xy} + \partial_y\sigma_{yy} + f_y \tag{5.73}$$

$$\begin{aligned}\partial_t\sigma_{xx} =& (\lambda + 2\mu)\partial_x v_x + \lambda\partial_y v_y + \frac{\mu+\lambda}{Q_1 + \sum_{l=1}^{L} D_{1,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} \\ &+ \frac{\mu}{Q_2 + \sum_{l=1}^{L} D_{2,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l}\end{aligned} \tag{5.74}$$

$$\begin{aligned}\partial_t\sigma_{yy} =& \lambda\partial_x v_x + (\lambda + 2\mu)\partial_y v_y + \frac{\mu+\lambda}{Q_1 + \sum_{l=1}^{L} D_{1,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} \\ &- \frac{\mu}{Q_2 + \sum_{l=1}^{L} D_{2,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l}\end{aligned} \tag{5.75}$$

$$\partial_t\sigma_{xy} = \mu(\partial_x v_y + \partial_y v_x) + \frac{\mu}{Q_2 + \sum_{l=1}^{L} D_{2,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xy,l} \tag{5.76}$$

$$\partial_t R_{1,l} = -R_{1,l}\theta_{1,l} - \theta_{1,l}D_{1,l}(\partial_x v_x + \partial_y v_y) \tag{5.77}$$

$$\partial_t R_{xx,l} = -R_{xx,l}\theta_{2,l} - \theta_{2,l}D_{2,l}(\partial_x v_x - \partial_y v_y) \tag{5.78}$$

$$\partial_t R_{xy,l} = -R_{xy,l}\theta_{2,l} - \theta_{2,l}D_{2,l}(\partial_x v_y + \partial_y v_x) \tag{5.79}$$

We recall that equations (5.77)–(5.79) are each N_{SLS} equations. Besides, we have omitted the piecewise linear improvement of the Q factor as described in section 5.3.3 to simplify readability.

5.3.5 Boundary conditions

When simulating wave propagation in real media as we are looking at in this thesis, the domain is only a section from a certain area, so there are no boundary conditions. However, the propagating waves decay only slowly, so that simply truncating the domain with hard-wall (Dirichlet or Neumann) or periodic boundary conditions leads to unacceptable artifacts of boundary reflections.

Therefore it is necessary to define reflection-free conditions at these boundaries to mimic an unlimited medium. For this purpose, we use a so-called perfectly matched layer

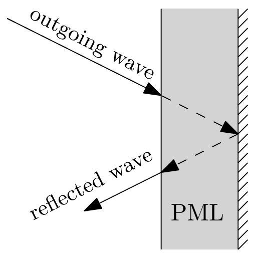


Figure 5.15: Wave absorption mechanism in a perfectly matched layer.

(PML) condition or more precisely the convolutional PML (CPML) condition, which is better unified in a discretized form. Roughly speaking, the area is limited by a layer in which the waves are damped and at the boundary of this layer the damped wave is reflected (see figure 5.15). Since these boundary conditions have no direct effect on our modifications, we refer to [23] for details.

5.4 Waveform inversion

Full waveform inversion means that the observed seismograms (possibly filtered) are considered as the basic observables that we want to fit. We thus searches for the model that minimizes the mean squared difference between observed and synthetic seismograms. In other words, the goal is to find a structural model that can explain a larger portion of seismological records, and not simply the phase of a few seismic arrivals.

We want to minimize the classical waveform misfit function:

$$\chi\left(m\right)=\sum_{s=1}^{N}\sum_{r=1}^{M}\int_{0}^{T}\frac{1}{2}\|u(x_r,x_s,m;t)-u_0(x_r,x_s;t)\|_2^2\,dt \tag{5.80}$$

This functional quantifies the L^2 difference between the observed waveforms $u_0(x_r,x_s;t)$ at receivers x_r, $r=1,...,M$ produced by sources at x_s, $s=1,...,N$, and the corresponding synthetic seismograms $u(x_r,x_s,m;t)$ computed in model m.

Following [99], we minimize the misfit function (5.80) subject to the constraint that the synthetic displacement fields satisfies the seismic wave equations (5.72)-(5.79). Mathematically, this implies the PDE-constrained minimization of the function

$$\begin{aligned}\chi_{PDE}\left(m\right)=&\sum_{s=1}^{N}\sum_{r=1}^{M}\int_{0}^{T}\frac{1}{2}\|u(x_r,x_s,m;t)-u_0(x_r,x_s;t)\|_2^2\,dt\\&-\sum_{i=0}^{4+3N_{\mathrm{SLS}}}\int_{0}^{T}\int_{\Omega}\lambda_i\left(\text{eq.}(5.72+i)\right),\end{aligned} \tag{5.81}$$

where the Lagrange multipliers λ_i remain to be determined. The equations (5.79+1) –(5.79+3($N_{\mathrm{SLS}}-1$)) denote the corresponding equations to equation (5.77)–(5.79) to update the memory variables added for $N_{\mathrm{SLS}}>1$. Finding an optimum of this minimization problem efficiently requires the gradient of equation (5.81), but computing the gradient is generally very expensive. However, it is possible to obtain this gradient without computing the Jacobian matrix explicitly. The approach to determine the gradient without computing the Fréchet derivatives was introduced in nonlinear optimization by [42], and later applied to seismic exploration problems [20]. The idea is to resort to the adjoint state, which corresponds to the wave field emitted and back-propagated from the receivers [130].

5.4.1 Regularization

Inverse problems are usually ill-posed, so a regularization term is often added to provide a more stable convergence, especially concerning noisy input data. Furthermore, the complete waveform inversion of a viscoelastic wave equation is much more complex than the purely elastic equation, which is more complex than for example an acoustic wave equation. Therefore, one key question is, which is the best regularization to achieve

better results and at least convergence to the true model.
A general regularization was introduced by Tikhonov [79, 148]. The starting point is that the data received at the receivers is disturbed. That means we search for an $u(m)$ for which $u_0 = u(m) - e$ for some random noise e. For a Gaussian distributed random noise, the estimation problem usually appears as determination of the minimizer of a suitably defined objective function

$$\chi(m)_{\text{reg}} = \chi(m) + \lambda_{\mathcal{R}}\mathcal{R}(m), \tag{5.82}$$

where $\chi(m)$ is given by equation (5.80), $\lambda_{\mathcal{R}}$ is a regularization weight and $\mathcal{R}$ is a regularizer or regularization function, which maps from the space of all model parameters into the real numbers and somehow prevents data overfitting.

Common regularization functions are given by:

$$\begin{aligned} \mathcal{R}_0(m) &= \|m\|_2^2 \\ \mathcal{R}_1(m) &= \|\nabla m\|_2^2 \\ \mathcal{R}_2(m) &= \|\Delta m\|_2^2 \end{aligned}$$

A good regularization usually describes properties that are expected of the solution, or that are known to be expected of the solution. For example, $\mathcal{R}_3$ describes a certain smoothness of the model m, or $\mathcal{R}_2$ is suitable for preventing oscillations and prefers plane surfaces and smooth variations.

Geophysical models usually have both sharp interfaces and smooth variations, and it is difficult to consider both types accurately. A possible regularization term that preserves both but still filters out oscillations and noise is a total variation denoising term

$$\mathcal{R}_{\text{TV}}(m) = \|m\|_{\text{TV}},$$

with the discrete TV norm in 2D is defined as

$$\|m\|_{\text{TV}} = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y}\sqrt{|(\nabla_x m)_{i,j}|^2 + |(\nabla_y m)_{i,j}|^2},$$

where N_x and N_y are the number of discrete points in x- and y-direction, respectively and $(\cdot)_{i.j}$ denotes the evaluation at point (i, j). Further advancement of TV regularization is the so-called total generalized p-variation regularization, which Gao and Huang introduced for acoustic and elastic waveform inversion [58] and which yields better results than standard TV regularization. This regularization results from the solution of another minimization problem and is described by

$$\mathcal{R}_{\mathcal{T}_p}(m) = \mathcal{T}_p(m) \tag{5.83}$$

$$\mathcal{T}_p(m) = \min_w \{\alpha_0 \|\nabla m - w\|_p^p + \alpha_1 \|\boldsymbol{\varepsilon}(w)\|_p^p\} \qquad (0 < p < 1) \tag{5.84}$$

where $w = (w_x w_y)$ is an auxiliary variable and $\boldsymbol{\varepsilon}(\cdot)$ is the symmetric gradient

$$\boldsymbol{\varepsilon}(w) = \begin{bmatrix} \nabla_x w_x & \frac{1}{2}(\nabla_x w_y + \nabla_y w_x) \\ \frac{1}{2}(\nabla_x w_y + \nabla_y w_x) & \nabla_y w_y \end{bmatrix}$$

The advantage of the p -variation is that it penalizes both the first-order gradient and high-order gradients of a model. Thus it can effectively avoid the so-called staircase effect compared with the first-order total variation [58].

5.4.2 Inversion parameters

Several parameters describe a property of isotropic linear elastic materials and two arbitrary parameters of these fully describe the material. Certainly, it is possible to use a different set of moduli in the forward simulation than for determining the gradients. Our equations are based for example on the parameters ρ, μ and λ. Another common parameter set is to use the pressure and shear velocities (v_p and v_s) instead of the Lamé parameters (μ and λ).

5.4.3 Adjoint state method

A classic technique to solve a non-linear minimization problem is to successively determine the minimum of a series of linearized problems. This formulation requires the Fréchet derivatives (the Jacobian matrix), which can be expensive to compute. If the minimization is viewed as a non-linear optimization problem, only the gradient of the error functional is needed. This gradient can be effectively computed without the Fréchet derivatives by using the adjoint state method. In the following, we derive the adjoint state method for the minimization problem (5.80). For the sake of clarity, we do not include the sums for the sources and recipients in the objective function.

The Fréchet derivative of the misfit $\chi(m)$ is the infinitesimal change of $\chi(m)$ as we pass from Earth model $m(x)$ to $m(x) + \delta m(x)$:

$$\frac{\mathrm{d}\chi}{\mathrm{d}m}\delta m := \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \left[\chi(m + \varepsilon \delta m) - \chi(m)\right]$$

The classical misfit functional in full waveform inversion is (see equation (5.80))

$$\chi = \frac{1}{2} \int \|u(x_r, m; t) - u_0(x_r; t)\|_2^2 \, \mathrm{d}t.$$

Using the Dirac delta function this can be rewritten as an integral over both time and space as

$$\chi = \frac{1}{2}\int\int \|u(x,m;t) - u_0(x;t)\|_2^2\,\delta(x-x_r)\mathrm{d}t\mathrm{d}x.$$

For a better readability we write in the following only the function names u and u_0 and omit their arguments. The Fréchet derivative of the misfit functional is then given by

$$\frac{\mathrm{d}\chi}{\mathrm{d}m}\delta m = \int\int \delta u \cdot \underbrace{(u-u_0)\delta(x-x_r)}_{=f^*}\mathrm{d}t\mathrm{d}x,$$

with $\delta u = \frac{\mathrm{d}u}{\mathrm{d}m}\delta m$. Besides the Misfit function, there are also constraints, which are given as a system of PDEs. We denote this system as $\mathbf{L} = f$, where $\mathbf{L}$ in our case describes the viscoelastic wave equation. Computing the Fréchet derivative of $\mathbf{L}$ in the direction δm yields

$$\frac{\mathrm{d}\mathbf{L}(u,m)}{\mathrm{d}u}\delta u + \frac{\mathrm{d}\mathbf{L}(u,m)}{\mathrm{d}m}\delta m = 0. \tag{5.85}$$

We can simplify the first term when $\mathbf{L}$ is linear in u to

$$\begin{aligned}\frac{\mathrm{d}\mathbf{L}(u,m)}{\mathrm{d}u}\delta u &= -\lim_{\varepsilon\to 0}\frac{1}{\varepsilon}\left[\mathbf{L}(u) - \mathbf{L}(u+\varepsilon\delta u)\right]\\ &= \lim_{\varepsilon\to 0}\frac{1}{\varepsilon}\left[\mathbf{L}(u+\varepsilon\delta u) - \mathbf{L}(u)\right]\\ &= \lim_{\varepsilon\to 0}\frac{1}{\varepsilon}\left[\mathbf{L}(u) + \varepsilon\mathbf{L}(\delta u) - \mathbf{L}(u)\right] = \mathbf{L}(\delta u).\end{aligned}$$

Thus equation (5.85) reads

$$\mathbf{L}(\delta u) + \frac{d\mathbf{L}(u,m)}{\mathrm{d}m}\delta m = 0.$$

Multiplying this with an arbitrary differentiable test function u^*, and integrating over time and space leads to

$$\int\int u^* \cdot \left[\mathbf{L}(\delta u) + \frac{d\mathbf{L}(u,m)}{\mathrm{d}m}\delta m\right]\mathrm{d}t\mathrm{d}x = 0. \tag{5.86}$$

Adding equation (5.86) to the Fréchet derivative of the misfit

$$\frac{d\chi}{\mathrm{d}m}\delta m = \int\int \delta u \cdot f^*\,\mathrm{d}t\mathrm{d}x$$

results in

$$\frac{\mathrm{d}\chi}{\mathrm{d}m}\delta m = \int\int \delta u \cdot f^* + u^* \cdot \mathbf{L}(\delta u) + u^* \cdot \frac{d\mathbf{L}(u,m)}{dm}\delta m\,\mathrm{d}t\mathrm{d}x \tag{5.87}$$

We eliminate δu from equation (5.87) with the help of adjoint of $\mathbf{L}$ that fulfills:

$$\int\int u^* \cdot \mathbf{L}(\delta u) dtdy = \int\int \delta u \cdot \mathbf{L}^*(u^*)\,\mathrm{d}t\mathrm{d}x$$

Finding the adjoints is the actual challenge, but if we manage to do so, we can transform equation (5.87) to:

$$\frac{\mathrm{d}\chi}{\mathrm{d}m}\delta m = \int\int \delta u \cdot [f^* + \mathbf{L}^*(u^*)] + u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}m}\delta m\,\mathrm{d}t\mathrm{d}x$$

We can eliminate δu when the adjoint field u^* satisfies the adjoint equation

$$\mathbf{L}^*(u^*) = -\,f^*.$$

All in all, computing the Fréchet derivative of the misfit χ then becomes quite easy and can be written as

$$\frac{\mathrm{d}\chi}{\mathrm{d}m}\delta m = \int\int u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}m}\delta m\,\mathrm{d}t\mathrm{d}x. \tag{5.88}$$

That means, before we can calculate the derivative of the misfit function, we first have to derive the adjoint model in the next section.

5.4.4 Adjoint model

In order to derive the adjoint model, we have to represent the forward model in matrix notation.

Forward model with matrix operations To derive the adjoint problem we need a mapping $\mathbf{L}$ describing the viscoelastic wave equation. For this purpose we write equations (5.72)–(5.79) in matrix notation. Therefore, we define the model parameters as a vector $m = (\lambda\ \mu\ \rho\ Q_1\ Q_2)^\top$, the simulation parameter vector is $u = (\sigma_{xx}\ \sigma_{yy}\ \sigma_{xy}\ v_x\ v_y\ R_{1,l}\ R_{xx,l}$ $R_{xy,l})^\top$ and the source term vector is $s = (0\ 0\ 0\ f_x\ f_y\ 0\ 0\ 0)^\top$. Thus the forward problem can be written as $\overline{\mathbf{L}}(u,m) = s$ or $\mathbf{L}(u,m) = 0$ with

$$\begin{aligned}\overline{\mathbf{L}}(u,m) &:= (-\partial_t I + A \cdot D + B)\,u,\\ \mathbf{L}(u,m) &:= \left(-A^{-1}\partial_t I + D + A^{-1}B\right)u + A^{-1}s.\end{aligned}$$

In the following, we consider the formulation $\mathbf{L}(u,m) = 0$ since the location-dependent model parameters do not appear in front of terms with spatial derivatives. This simplifies the partial integration when deriving the adjoint problem. Here the matrices A, B and

D are given by

$$A = \begin{pmatrix} (\lambda+2\mu) & \lambda & 0 & 0 & 0 & 0 & 0 & 0 \\ \lambda & (\lambda+2\mu) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\rho} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\rho} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -D_{1,l}\theta_{1,l} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -D_{2,l}\theta_{2,l} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -D_{2,l}\theta_{2,l} \end{pmatrix},$$

$$D = \begin{pmatrix} 0 & 0 & 0 & \partial_x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \partial_y & 0 & 0 & 0 \\ 0 & 0 & 0 & \partial_y & \partial_x & 0 & 0 & 0 \\ \partial_x & 0 & \partial_y & 0 & 0 & 0 & 0 & 0 \\ 0 & \partial_y & \partial_x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \partial_x & \partial_y & 0 & 0 & 0 \\ 0 & 0 & 0 & \partial_x & -\partial_y & 0 & 0 & 0 \\ 0 & 0 & 0 & \partial_y & \partial_x & 0 & 0 & 0 \end{pmatrix} \text{ and}$$

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & \frac{(\lambda+\mu)}{Q_1+\sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}} & \frac{\mu}{Q_2+\sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{(\lambda+\mu)}{Q_1+\sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}} & -\frac{\mu}{Q_2+\sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\mu}{Q_2+\sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\theta_{1,l} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\theta_{2,l} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\theta_{2,l} \end{pmatrix}.$$

Derivation of adjoint model Using the matrix notation we can now determine the adjoint problem. For this we use the relation between forward and adjoint problem

$$\int\int u^* \cdot \mathbf{L}(\delta u)\,\mathrm{d}t\mathrm{d}x = \int\int \delta u \cdot \mathbf{L}^*(u^*)\,\mathrm{d}t\mathrm{d}x$$

For better readability we write both sides as bilinear forms which reads

$$\langle u^*, \mathbf{L}(\delta u)\rangle = \langle \delta u, \mathbf{L}^*(u^*)\rangle\,.$$

Inserting $\mathbf{L}(\delta u)$ into the left side yields

$$\langle u^*, \mathbf{L}(\delta u)\rangle = \left\langle u^*, (-A^{-1}\partial_t I + D + A^{-1}B)\delta u \right\rangle .$$

We split the bilinear form into three parts and consider each term individually:

$$\begin{aligned}
&= \underbrace{\left\langle u^*, -A^{-1}\partial_t I\delta u\right\rangle}_{=I} + \underbrace{\langle u^*, D\delta u\rangle}_{=II} + \underbrace{\left\langle u^*, A^{-1}B\delta u\right\rangle}_{=III} \\
I &= \left\langle u^*, -A^{-1}\partial_t I\delta u\right\rangle \overbrace{=}^{\text{P.I.}} - \left\langle (-A^{-1})^\top \partial_t I u^*, \delta u\right\rangle \\
II &= \langle u^*, D\delta u\rangle \overbrace{=}^{\text{P.I.}} - \left\langle D^\top u^*, \delta u\right\rangle \\
III &= \left\langle u^*, A^{-1}B\delta u\right\rangle = \left\langle B^\top (A^{-1})^\top u^*, \delta u\right\rangle
\end{aligned}$$

In each term the property $< M\ \cdot, \cdot >=< \cdot, M^\top\ \cdot >$ for $M \in \mathbb{R}^{n\times n}$ of the bilinear form was used. We had also performed a partial integration in I and II.

Next, we combine all three terms and since A is symmetric, this leads to

$$\langle u^*, \mathbf{L}(\delta u)\rangle = \left\langle -(-A^{-1}\partial_t I + D^\top - (A^{-1}B)^\top)u^*, \delta u\right\rangle .$$

Due to the symmetry of this bilinear form, we can identify the adjoint problem, that is

$$\mathbf{L}^*(u^*) = -(-A^{-1}\partial_t I + D^\top - (A^{-1}B)^\top)u^*. \tag{5.89}$$

For better comparability of the adjoint problem with the forward problem, we rewrite the matrix notation into a system of equations. Therefore, we first calculate $(A^{-1}B)^\top$ and get

$$\left(A^{-1}B\right)^\top = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2}\frac{1}{Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l}} & \frac{1}{2}\frac{1}{Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l}} & 0 & 0 & 0 & \frac{1}{D_{1,l}} & 0 & 0 \\
\frac{1}{2}\frac{1}{Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l}} & -\frac{1}{2}\frac{1}{Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l}} & 0 & 0 & 0 & 0 & \frac{1}{D_{2,l}} & 0 \\
0 & 0 & \frac{1}{Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l}} & 0 & 0 & 0 & 0 & \frac{1}{D_{2,l}}
\end{pmatrix}$$

All in all, the adjoint problem $A \cdot \mathbf{L}^*(u^*) = 0$ is given by the following equations:

$$\partial_t v_x^* = \frac{1}{\rho}(\partial_x \sigma_{xx}^* + \partial_y \sigma_{xy}^* + \sum_{l=1}^{N_{\mathrm{SLS}}} (\partial_x R_{1,l}^* + \partial_x R_{xx,l}^* + \partial_y R_{xy,l}^*)) \tag{5.90}$$

$$\partial_t v_y^* = \frac{1}{\rho}(\partial_x \sigma_{xy}^* + \partial_y \sigma_{yy}^* + \sum_{l=1}^{N_{\mathrm{SLS}}} (\partial_y R_{1,l}^* - \partial_y R_{xx,l}^* + \partial_x R_{xy,l}^*)) \tag{5.91}$$

$$\partial_t \sigma_{xx}^* = (\lambda + 2\mu)\partial_x v_x^* + \lambda \partial_y v_y^* \tag{5.92}$$

$$\partial_t \sigma_{yy}^* = \lambda \partial_x v_x^* + (\lambda + 2\mu)\partial_y v_y^* \tag{5.93}$$

$$\partial_t \sigma_{xy}^* = \mu(\partial_x v_y^* + \partial_y v_x^*) \tag{5.94}$$

$$\partial_t R_{1,l}^* = \frac{1}{2}\frac{D_{1,l}\theta_{1,l}}{Q_1 + \sum D_{1,l}} \left(\sigma_{xx}^* + \sigma_{yy}^*\right) + \theta_{1,l} R_{1,l}^* \tag{5.95}$$

$$\partial_t R_{xx,l}^* = \frac{1}{2}\frac{D_{2,l}\theta_{2,l}}{Q_2 + \sum D_{2,l}} \left(\sigma_{xx}^* - \sigma_{yy}^*\right) + \theta_{2,l} R_{xx,l}^* \tag{5.96}$$

$$\partial_t R_{xy,l}^* = \frac{D_{2,l}\theta_{2l,}}{Q_2 + \sum D_{2,l}} \sigma_{xy}^* + \theta_{2,l} R_{xy,l}^* \tag{5.97}$$

In contrast to the pure elastic wave equation, where the adjoint problem is the same as the forward problem, the adjoint viscoelastic wave equation differs especially in the equations for the velocity update.

5.4.5 Derivative of the misfit function

After we have derived the adjoint problem, we can now use equation (5.88) to determine the derivative of the misfit function.

First we calculate the derivative for the linear operator of the forward problem

$$\mathbf{L}(u, m) = -A^{-1}\partial_t u + Du + A^{-1}Bu + A^{-1}s,$$

which is

$$\begin{aligned}
\frac{d\mathbf{L}(u,m)}{\mathrm{d}m} &= -\partial_m A^{-1}\partial_t u + \partial_m \left(A^{-1}B\right) u + \partial_m A^{-1} s \\
&= -\partial_m A^{-1}\partial_t u + \partial_m A^{-1}Bu + A^{-1}\partial_m Bu + \partial_m A^{-1} s \\
&= \partial_m A^{-1} \underbrace{(-\partial_t u + Bu + s)}_{=-ADu} + A^{-1}\partial_m Bu \\
&= -\partial_m A^{-1}(AD)u + A^{-1}\partial_m Bu
\end{aligned}$$

Next, we can insert this derivative into equation (5.88) and obtain

$$\frac{\mathrm{d}\chi}{\mathrm{d}m}\delta m = \int\int u^* \cdot \left(-\partial_m A^{-1}(AD) + A^{-1}\partial_m B\right) u \;\mathrm{d}x\mathrm{d}t. \tag{5.98}$$

Now we are able to calculate the derivatives. We omit here the detailed calculation of the derivatives, which are performed in section A.1.4 in the appendix. The following derivations result:

$$\begin{aligned}
u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}\rho} &= u^* \cdot (-\partial_\rho A^{-1}(AD) + A^{-1}\partial_\rho B)u \\
&= -v_x^* \partial_t \tilde{v}_x - v_y^* \partial_t \tilde{v}_y.
\end{aligned}$$

$$\begin{aligned}
u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}\lambda} &= u^* \cdot (-\partial_\lambda A^{-1}(AD) + A^{-1}\partial_\lambda B)u \\
=& \frac{1}{4(\lambda+\mu)^2} \left\{ (\sigma_{xx}^* + \sigma_{yy}^*)(\partial_t \tilde{\sigma}_{xx} + \partial_t \tilde{\sigma}_{yy}) \right\} \\
&+ (\sigma_{xx}^* + \sigma_{yy}^*) \frac{1}{2(\lambda+\mu)} \frac{1}{Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l}
\end{aligned}$$

$$\begin{aligned}
u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}\mu} =& u^* \cdot (-\partial_\mu A^{-1}(AD) + A^{-1}\partial_\mu B)u \\
=& \sigma_{xx}^* \left(\frac{\lambda^2 + 2\lambda\mu + 2\mu^2}{4\mu^2(\lambda+\mu)^2} \partial_t \tilde{\sigma}_{xx} - \frac{\lambda(\lambda+2\mu)}{4\mu^2(\lambda+\mu)^2} \partial_t \tilde{\sigma}_{yy} \right) \\
&- \sigma_{yy}^* \left(\frac{\lambda(\lambda+2\mu)}{4\mu^2(\lambda+\mu)^2} \partial_t \tilde{\sigma}_{xx} - \frac{\lambda^2 + 2\lambda\mu + 2\mu^2}{4\mu^2(\lambda+\mu)^2} \partial_t \tilde{\sigma}_{yy} \right) + \sigma_{xy}^* \left(\frac{1}{\mu^2} \partial_t \tilde{\sigma}_{xy} \right) \\
&+ (\sigma_{xx}^* + \sigma_{yy}^*) \frac{1}{2(\lambda+\mu)} \frac{1}{Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} \\
&+ (\sigma_{xx}^* - \sigma_{yy}^*) \frac{1}{2(\mu)} \frac{1}{Q_2 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} \\
&+ (\sigma_{xy}^*) \frac{1}{\mu} \frac{1}{Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}} \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xy,l}
\end{aligned}$$

$$\begin{aligned}
u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}Q_1} =& u^* \cdot (-\partial_{Q_1} A^{-1}(AD) + A^{-1}\partial_{Q_1} B)u \\
=& -\frac{1}{2} \frac{1}{\left(Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}\right)^2} \left(\sigma_{xx}^* \sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} + \sigma_{yy}^* \sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} \right)
\end{aligned}$$

$$\begin{aligned}
u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}Q_2} =& u^* \cdot (-\partial_{Q_2} A^{-1}(AD) + A^{-1}\partial_{Q_2} B)u \\
=& \frac{1}{2} \frac{1}{\left(Q_2 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}\right)^2} \left(\sigma_{yy}^* \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} - \sigma_{xx}^* \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} - 2\sigma_{xy}^* \sum_{l=1}^{N_{\mathrm{SLS}}} R_{xy,l} \right)
\end{aligned}$$

All in all, with the forward, adjoint equations and the partial derivatives of the misfit function, we now have everything we need to apply an optimization algorithm to the minimization problem (5.80). The implementation that is used in this thesis is presented in the following section.

5.5 Algorithmic realization

In this section, we present a possible realization of a full waveform inversion algorithm. We start with a simple version, where we restrict ourselves to only one source. Then, we discuss the components of this algorithm and improve this approach, e.g., with respect to parallelism and convergence, so that we finally arrive at the algorithm used in this thesis. The realizations presented here are part of the FWI simulation software SOUND-VIEW, with which our numerical tests in section 5.6 were accomplished and in which the contributions in this section were implemented.

The input data of each FWI algorithm are taken from a previous forward run, for example from a physical experiment. Thus, the input data of the algorithm are the source terms of the sources from the forward simulation as well as the signals collected at the receivers. In addition, an initial model is required, which the algorithm iteratively approximates to the target model.

Afterwards, an optimization algorithm is used to solve the minimization problem (5.80). As described in the previous section, we determine the derivative of the misfit function without directly calculating the Fréchet derivative. Instead, we use the adjoint state method. For this method, we need the solution of the forward and adjoint simulation. Within an iteration of the optimization algorithm, first, a forward simulation and then an adjoint simulation must be performed. Using these two solutions, the derivative of the misfit function can be determined and the optimization algorithm can calculate the next approximation of the model.

Algorithm 5.1 (simple FWI algorithm).

```
 1: Input: initial model $m_0$, source signals $\mathbf{s}$, receiver signals $\mathbf{r}$
 2: Output: model $m$
 3: $i = 1$
 4: while $|m_i - m_{i-1}| < TOL$ do
 5:     forward simulation                                  ▷ section 5.3.4
 6:     adjoint simulation                                  ▷ section 5.4.4
 7:     calculate derivative of misfit function             ▷ section 5.4.5
 8:     update of optimization method($m_{i+1}$, $m_i$)
 9:     $i = i + 1$
10: end while
```

The basic/simplified algorithm is shown in algorithm 5.1. In the following sections, we discuss the optimization procedure and the implementation of the forward and adjoint simulation.

5.5.1 Optimization method

To obtain a solution of the quadratic minimization problem (5.80), we use an iterative method described by Broyden, Fletcher, Goldfarb, and Shanno [38, 55, 62, 138] and named after as BFGS method. Considering the viscoelastic PDE instead of the elastic one increases the number of inverting parameters, but the modifications from the sections 5.3 and 5.4 do not affect the l-BFGS approach.

Limited-memory BFGS

The line search method is a basic method to solve an optimization problem iteratively. Here a search direction p_k and a step size s_k is determined, with which the next iteration is calculated as

$$m_{k+1} = m_k + s_k p_k.$$

Using iterative methods, it is possible to compute an estimate of the inverse Hessian based only on the knowledge of the gradient at the previous iterations, the quality of the approximation improving with the number of previous iterations used. The method generates a series of models that gradually converge towards a minimum of the misfit function (which may be local), and a series of matrices that converge towards the inverse Hessian.

A typical choice is that p_k is determined as the negative gradient

$$p_k = -\nabla\chi\,,$$

The BFGS formula to compute H_k, the approximate inverse Hessian at iteration k, is given by [119]:

$$H_k^{-1} \simeq H_{k-1}^{-1} - \frac{H_{k-1}^{-1} \cdot s_{k-1} \otimes s_{k-1} \cdot H_{k-1}^{-1}}{s_{k-1} \cdot H_{k-1} \cdot s_{k-1}} + \frac{y_{k-1} \otimes y_{k-1}}{y_{k-1} \cdot s_{k-1}}\,, \tag{5.99}$$

where $\otimes$ is the tensor product, $s_k = m_k - m_{k-1}$ is the difference between the current model and the model at the previous iteration, and $y_k = \nabla\chi_k - \nabla\chi_{k-1}$ is the gradient change. Using equation (5.99) we can iteratively calculate an estimate of the inverse Hessian H^{-1} based on the knowledge of the approximation of the inverse Hessian at the previous iteration, the difference s between the current model and its value at the previous iteration, and the difference y between the current gradient and the gradient at the previous iteration. Compared to the gradient method, convergence of BFGS is much faster for the same numerical cost [119]. Compared to the classical Gauss-Newton method, BFGS is also easier to implement because it does not require to compute and store the inverse Hessian. The reader is referred to [119] for a more detailed presentation of the BFGS algorithm.

To compute the search direction at iteration k

$$p_k = -H_k^{-1} \cdot \nabla\chi \,, \tag{5.100}$$

we only needs to perform a matrix-vector multiplication. However, in the case of large problems it is currently impossible to compute and store even the approximate inverse Hessian matrix. Since in equation (5.100) we do not need to explicitly store it but only be able to compute its effect on a vector (the gradient), a modified method called the l-BFGS algorithm (for 'limited-memory BFGS') has been developed [38, 55, 62, 138] in order to compute the matrix-vector product in equation (5.100) without having to store the inverse Hessian. The principle of l-BFGS is to use equation (5.99) iteratively to compute the product of the inverse Hessian, using the gradient from the initial inverse Hessian and the history of models and gradients accumulated in the iterations of the algorithm. In this case, we only need to store a set of models and gradients, which represents only a fraction of the storage space required to store the complete inverse Hessian. The number of previous models and gradients that are kept in memory is a parameter chosen by the user, e.g., we use $l = 20$ in our simulations in section 5.6.

Calculation of the step length Once the descent direction p_k at iteration k has been obtained, it is necessary to determine the step length, or in other words to decide how far to move along that direction. This problem can be formulated as finding the step s that minimizes

$$\phi(s) = \chi(m_k + s p_k) \,.$$

In practice, determining that optimal step precisely may require to test a large number of step lengths, which can thus be very expensive. However, we should keep in mind that $\chi(m)$ rather than $\phi(s)$ is the quantity that we need to minimize. It is thus sufficient to find an approximate step at minimal cost that honors certain conditions in order to make the optimization method converge. In practice, the step length variations between two iterations must be sufficiently large so that the algorithm requires a moderate number of iterations to converge, and sufficiently small to avoid the divergence of the algorithm. A good compromise is to use the so-called Wolfe conditions to select the step length [166, 167]. These rules test if the current step provides a sufficient decrease of both the cost function and the gradient. Introducing parameters $0 < c_1 < c_2 < 1$, and $\phi'(s)$ the derivative of ϕ with respect to s, the step length is kept if

$$\phi(s) \leq \phi(0) + c_1 s \phi'(0) \quad \text{and} \quad |\phi'(s)| \leq c_2 |\phi'(0)| \,.$$

If these two conditions are not satisfied, a new step is tested. If

$$\phi(s) > \phi(0) + c_1 s \phi'(0)$$

the step is too large, and we then tests a smaller step length. On the other hand, if

$$\phi(s) \leq \phi(0) + c_1 s \phi'(0) \quad \text{and} \quad |\phi'(s)| > c_2 |\phi'(0)| \,,$$

the step is too short, and we then test a longer step. When it is no longer possible to find a step that satisfies these relations, convergence is reached and we then stop the algorithm. However, this can also mean that a local minimum and not the global minimum has been reached (cf. section 5.5.3). Tuning parameters c_1 and c_2 makes the selection rules more or less restrictive in terms of accepting the step length. For example, if c_1 is chosen close to 0, it is easier to honor the first inequality. In our implementation, we select $c_1 = 0.1$ and $c_2 = 0.9$, the standard values recommended by [119].

Alternating L_2-TV optimization

The gradient of the misfit function is calculated using the adjoint state method (see section 5.4.5). If an additional regularization procedure is used, the gradient can be calculated additively. However, if we use a total variation type regularization like

$$\chi_{\mathrm{TV}}(m) = \min_m \left\{ \sum_{s=1}^{N} \sum_{r=1}^{M} \int_0^T \frac{1}{2} \| u(x_r, x_s, m; t) - u_0(x_r, x_s; t) \|_2^2 \, dt + \lambda_{\mathcal{R}_{\mathrm{TV}}} \| m \|_{\mathrm{TV}} \right\},$$

an alternating approach gives better results [63]. For better readability, we omit the sums for the sources and recipients in the following. To achieve an alternating method we introduce an auxiliary variable v and use the equivalent minimization problem

$$\chi_{\mathrm{TV,alt}}(m, v) = \min_{m,v} \left\{ \int_0^T \frac{1}{2} \| u(m; t) - u_0(t) \|_2^2 \, dt + \lambda_{\mathrm{TV},1} \| m - v \|_2^2 + \lambda_{\mathrm{TV},2} \| v \|_{\mathrm{TV}} \right\},$$

where $\lambda_{\mathrm{TV},1}$ and $\lambda_{\mathrm{TV},2}$ are regularization parameters. To solve this minimization problem we use an alternating minimization algorithm, this leads to two independent minimization problems

$$m^{(k)} = \min_m \chi_{\mathrm{TV,alt1}}(m) = \min_m \left\{ \int_0^T \frac{1}{2} \| u(m; t) - u_0(t) \|_2^2 \, dt + \lambda_{\mathrm{TV},1} \| m - v^{(k-1)} \|_2^2 \right\}$$

and

$$v^{(k)} = \min_v \chi_{\mathrm{TV,alt2}}(u) = \min_v \left\{ \lambda_{\mathrm{TV},1} \| m^{(k)} - v \|_2^2 + \lambda_{\mathrm{TV},2} \| v \|_{\mathrm{TV}} \right\}$$

A further improvement of this alternating method is the use of so-called alternating direction method multipliers (ADMM) [30], which result in the following minimization problems

$$\begin{aligned}
m^{(k)} &= \min_m \left\{ \int_0^T \frac{1}{2} \|u(m;t) - u_0(t)\|_2^2 \, dt + \lambda_{\mathrm{TV},1} \|m - v^{(k-1)} + q^{(k-1)}\|_2^2 \right\} \\
v^{(k)} &= \min_u \left\{ \lambda_{\mathrm{TV},1} \|m^{(k)} - v + q^{(k-1)}\|_2^2 + \lambda_{\mathrm{TV},2} \|v\|_{\mathrm{TV}} \right\} \\
q^{(k)} &= q^{(k-1)} + m^{(k)} - v^{(k)}
\end{aligned}$$

To solve the first minimization problem we use the l-BFGS method and for the second minimization problem, we use the split-Bregman method. We will skip details about the split-Bregman method here and refer to a work by Goldstein and Osher [63]. The p-variation regularization can be implemented in the same way. A detailed description of such an realization and algorithmic implementation is described in the appendix in section A.1.6.

5.5.2 Forward/adjoint simulation

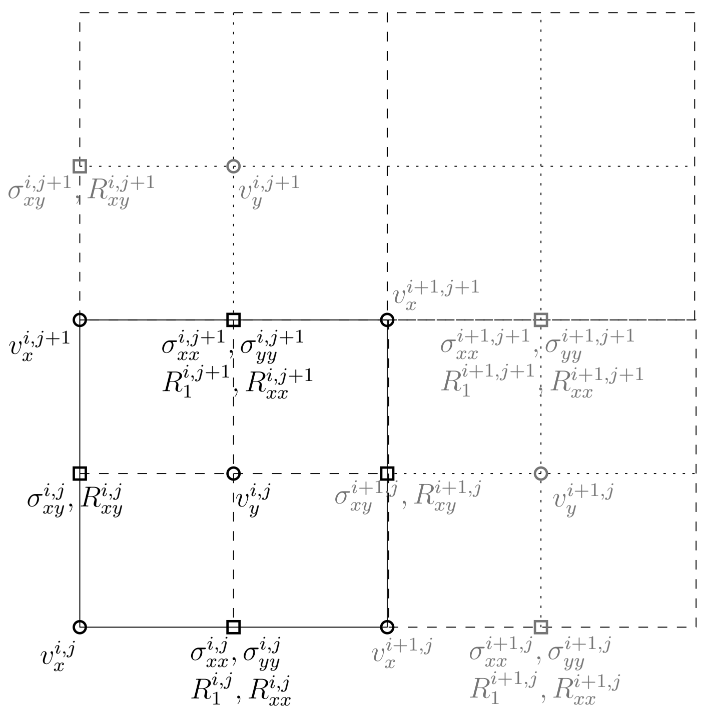
Staggered grid To discretize the forward and adjoint simulation we use a so-called staggered grid finite difference approach. In this approach, not all parameters are given in the same degrees of freedom but are shifted by half a step size. The staggered grid definition used in this thesis is shown in figure 5.16. It is important to note that the parameters which are calculated live in the same degree of freedom and are interpolated in this way. The main benefit of the staggered grid over collocated grids is that, unlike the collocated grid, the staggered grid does not results in the decoupling of the pressure and velocity, leading to the checkerboard problem. To decouple the equations in time, we use an alternating approach. This means that the velocity update is done in time step $n + 1/2$ and the stress update in full time step $n + 1$.

A balanced direct discretization is achieved if a higher domain discretization is chosen as time discretization. Blanch et al. [27] have shown that order four in space and order two in time is a good choice.

In the following, we discretize the equations exemplarily with the Crank-Nicolson method in time and a central second order coefficient in domain (to achieve better readability than it would be the case using the fourth order discretization).

Spatial discretization Any finite difference approach can be used for the spatial discretization. For example, we use a central second order difference quotient. Further, we use superscripts i, j to mark the respective grid points. Figure 5.16 shows that identical indices of different components and parameters are not necessarily in the same position. So it is necessary to make sure that the parameters are interpolated so that all calculated parameters describe the same grid position.

We further mark the discrete evaluated derivative at point i, j with a small h in the subscript so that $\partial_{x,h} v_x^{i,j}$ describes the numerical approximation of the derivative evaluated



$\sigma_{xy}^{i,j+1}, R_{xy}^{i,j+1}$
$v_y^{i,j+1}$
$v_x^{i+1,j+1}$
$v_x^{i,j+1}$
$\sigma_{xx}^{i,j+1}, \sigma_{yy}^{i,j+1}$
$R_1^{i,j+1}, R_{xx}^{i,j+1}$
$\sigma_{xx}^{i+1,j+1}, \sigma_{yy}^{i+1,j+1}$
$R_1^{i+1,j+1}, R_{xx}^{i+1,j+1}$
$\sigma_{xy}^{i,j}, R_{xy}^{i,j}$
$v_y^{i,j}$
$\sigma_{xy}^{i+1,j}, R_{xy}^{i+1,j}$
$v_y^{i+1,j}$
$v_x^{i,j}$
$\sigma_{xx}^{i,j}, \sigma_{yy}^{i,j}$
$R_1^{i,j}, R_{xx}^{i,j}$
$v_x^{i+1,j}$
$\sigma_{xx}^{i+1,j}, \sigma_{yy}^{i+1,j}$
$R_1^{i+1,j}, R_{xx}^{i+1,j}$

Figure 5.16: Staggered grid for the viscoelastic wave equation. The stress and memory variables are defined in the points marked by squares and the velocities are defined in the points marked by circles.

at point i, j. Since the partial derivative $\partial_x v_x$ only occurs in the two equations for the determination of σ_{xx} and σ_{yy}, the derivative is also only needed in this point, so that $\partial_{x,h} v_x^{i,j}$ is only needed at the position from σ_{xx}. Thus the derivative at this point is given by the central difference quotient evaluated in $v_x^{i+1,j}$ and $v_x^{i,j}$ as shown in figure 5.17 on the left, and the derivative is given by

$$\partial_{x,h} v_x^{i,j} = (v_x^{i+1,j} - v_x^{i,j})/h.$$

We also see in figure 5.17 that this is really a central difference quotient, because the partial derivative $\partial_{x,h} v_x^{i,j}$ is calculated at a point that lies exactly between $v_x^{i+1,j}$ and $v_x^{i,j}$. In the same way, the remaining partial derivatives of the velocity are:

$$\begin{aligned}
\partial_{y,h} v_x^{i,j} &= (v_x^{i,j+1} - v_x^{i,j})/h \\
\partial_{x,h} v_y^{i,j} &= (v_y^{i,j} - v_y^{i-1,j})/h \\
\partial_{y,h} v_y^{i,j} &= (v_y^{i,j} - v_y^{i,j-1})/h
\end{aligned}$$

All partial derivatives are only needed in exactly one point of the staggered grid, these places are shown on the right side of figure 5.17 (here the superscript i, j was skipped, since it is the same everywhere).

Analogous to the partial derivatives of the velocity components, the derivatives of the

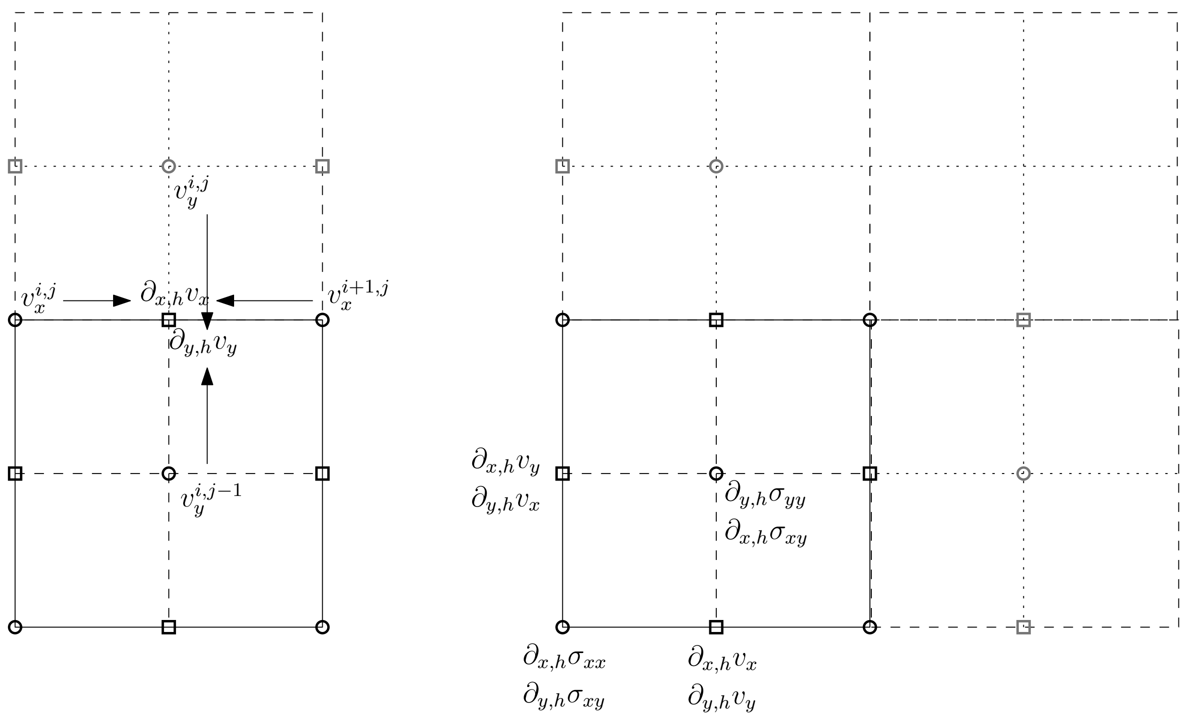


Figure 5.17: left: central difference quotient in a staggered grid; right: positions of partial derivatives in a staggered grid, whereas the positions are the same as in figure 5.16.

stress vector components can be determined, and are as follows:

$$\begin{aligned}
\partial_{x,h}\sigma_{xx}^{i,j} &= (\sigma_{xx}^{i,j} - \sigma_{xx}^{i-1,j})/h \\
\partial_{y,h}\sigma_{yy}^{i,j} &= (\sigma_{yy}^{i,j+1} - \sigma_{yy}^{i,j})/h \\
\partial_{x,h}\sigma_{xy}^{i,j} &= (\sigma_{xy}^{i,j} - \sigma_{xy}^{i,j-1})/h \\
\partial_{y,h}\sigma_{xy}^{i,j} &= (\sigma_{xy}^{i+1,j} - \sigma_{xy}^{i,j})/h
\end{aligned}$$

Compare again the figures 5.17 and 5.16 to see that these are central difference quotients of order 2.

Time discretization of the forward simulation After the spatial discretization has been done, we now perform time discretization. Since the time discretization has to be done for each discrete point in space, we do not use the location indices i, j in the following because all parameters are evaluated in the same point and we use the superscript n for the $n-$th time step instead.

We start with the equations of the forward simulation, insert the spatial discretizations in equations (5.72)–(5.79) and discretize them with the Crank-Nicolson method [46]. We recall that the velocity update is done in time step $n + 1/2$ and the stress update in full time step $n + 1$. Applying the Crank-Nicolson method to equation (5.72), we obtain

$$\rho \frac{v_x^{n+1/2} - v_x^{n-1/2}}{\Delta t} = \partial_{x,h}\sigma_{xx}^n + \partial_{y,h}\sigma_{xy}^n.$$

Elementary restructuring results in the update rule

$$v_x^{n+1/2} = v_x^{n-1/2} + \frac{\Delta t}{\rho}(\partial_{x,h}\sigma_{xx}^n + \partial_{y,h}\sigma_{xy}^n). \tag{5.101}$$

Similarly, this results for equation (5.73) in

$$v_y^{n+1/2} = v_y^{n-1/2} + \frac{\Delta t}{\rho}(\partial_{x,h}\sigma_{xy}^n + \partial_{y,h}\sigma_{yy}^n). \tag{5.102}$$

Since σ and the memory variables are defined in the same time step, using the Crank-Nicolson procedure for equation (5.74) is a bit more complex, and we get

$$\begin{aligned}\frac{\sigma_{xx}^{n+1} - \sigma_{xx}^n}{\Delta t} =&(\lambda + 2\mu)\partial_{x,h}v_x^{n+1/2} + \lambda\partial_{y,h}v_y^{n+1/2}\\ &+ \frac{1}{2}\left(\frac{\lambda+\mu}{Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l}^{n+1} + \frac{\mu}{Q_2 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l}^{n+1}\right.\\ &\left.+\frac{\lambda+\mu}{Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D1,l}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l}^{n} + \frac{\mu}{Q_2 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}}\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l}^{n}\right).\end{aligned}$$

Elementary restructuring and summarizing yields the update rule

$$\begin{aligned}\sigma_{xx}^{n+1} =&\sigma_{xx}^n + \Delta t\left((\lambda+2\mu)\partial_{x,h}v_x^{n+1/2} + \lambda\partial_{y,h}v_y^{n+1/2}\right)\\ &+ \frac{\Delta t}{2}\frac{\lambda+\mu}{Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}}\left(\sum_{l=1}^{N_{\mathrm{SLS}}}(R_{1,l}^{n+1} + R_{1,l}^n)\right)\\ &+ \frac{\Delta t}{2}\frac{\mu}{Q_2 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}}\left(\sum_{l=1}^{N_{\mathrm{SLS}}}(R_{xx,l}^{n+1} + R_{xx,l}^n)\right).\end{aligned} \tag{5.103}$$

In a similar way we obtain the update regulations for σ_{yy} und σ_{xy}:

$$\begin{aligned}\sigma_{yy}^{n+1} =&\sigma_{yy}^n + \Delta t\left(\lambda\partial_{x,h}v_x^{n+1/2} + (\lambda+2\mu)\partial_{y,h}v_y^{n+1/2}\right)\\ &+ \frac{\Delta t}{2}\frac{\lambda+\mu}{Q_1 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{1,l}}\left(\sum_{l=1}^{N_{\mathrm{SLS}}}(R_{1,l}^{n+1} + R_{1,l}^n)\right)\\ &- \frac{\Delta t}{2}\frac{\mu}{Q_2 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}}\left(\sum_{l=1}^{N_{\mathrm{SLS}}}(R_{xx,l}^{n+1} + R_{xx,l}^n)\right)\end{aligned} \tag{5.104}$$

$$\begin{aligned}\sigma_{xy}^{n+1} =&\sigma_{xy}^n + \Delta t\mu\left(\partial_{x,h}v_y^{n+1/2} + \partial_{y,h}v_x^{n+1/2}\right)\\ &+ \frac{\Delta t}{2}\frac{\mu}{Q_2 + \sum_{l=1}^{N_{\mathrm{SLS}}} D_{2,l}}\left(\sum_{l=1}^{N_{\mathrm{SLS}}}(R_{xy,l}^{n+1} + R_{xy,l}^n)\right)\end{aligned} \tag{5.105}$$

Finally the update rules for the memory variables are missing. The Crank-Nicolson procedure exemplary for equation (5.77) results in

$$\frac{R_{1,l}^{n+1} - R_{1,l}^{n}}{\Delta t} = \frac{1}{2}\left(-R_{1,l}^{n+1}\theta_{1,l} - R_{1,l}^{n}\theta_{1,l}\right) - \theta_{1,l}D_{1,l}\left(\partial_{x,h}v_x^{n+1/2} + \partial_{y,h}v_y^{n+1/2}\right).$$

Elementary restructuring and summarizing yields the update rules

$$R_{1,l}^{n+1} = \frac{1}{1+0.5\Delta t\theta_{1,l}}\left(R_{1,l}^{n} - \frac{\Delta t}{2}\theta_{1,l}R_{1,l}^{n} - \Delta t\theta_{1,l}D_{1,l}(\partial_{x,h}v_x^{n+1/2} + \partial_{y,h}v_y^{n+1/2})\right). \quad (5.106)$$

Similarly, the update rules for the other memory variables are:

$$R_{xx,l}^{n+1} = \frac{1}{1+0.5\Delta t\theta_{2,l}}\left(R_{xx,l}^{n} - \frac{\Delta t}{2}\theta_{2,l}R_{xx,l}^{n} - \Delta t\theta_{2,l}D_{2,l}(\partial_{x,h}v_x^{n+1/2} - \partial_{y,h}v_y^{n+1/2})\right) \quad (5.107)$$

$$R_{xy,l}^{n+1} = \frac{1}{1+0.5\Delta t\theta_{2,l}}\left(R_{xy,l}^{n} - \frac{\Delta t}{2}\theta_{2,l}R_{xy,l}^{n} - \Delta t\theta_{2,l}D_{2,l}(\partial_{x,h}v_y^{n+1/2} + \partial_{y,h}v_x^{n+1/2})\right) \quad (5.108)$$

Thus we have completely discretized the forward problem.

Time discretization of the adjoint simulation For the discretization of the adjoint simulation, the same discrete derivatives are used as for the forward simulation. We need additionally the partial derivatives of the memory variables in the adjoint problem. For better readability and since all variables in this section are the adjoint variables, we do not mark these quantities with a '$*$'. Since the memory variables are located in the same points as the corresponding stress components and occur in the same equations, they are calculated analogously, e.g., $\partial_{x,h}R_{xx,l}$ is calculated analogously to $\partial_{x,h}\sigma_{xx}$ (see figure 5.16 and 5.17).

We insert an arbitrary spatial discretization into the equations (5.90)–(5.97), discretize them with the Crank-Nicolson method and receive the following update rules in the same way as for the forward simulation:

$$v_x^{n+1/2} = v_x^{n-1/2} + \frac{\Delta t}{\rho}(\partial_{x,h}\sigma_{xx}^n + \partial_{y,h}\sigma_{xy}^n + \sum_{l=1}^{N_{\text{SLS}}}(\partial_{x,h}R_{1,l}^n + \partial_{x,h}R_{xx,l}^n + \partial_{y,h}R_{xy,l}^n)) \quad (5.109)$$

$$v_y^{n+1/2} = v_y^{n-1/2} + \frac{\Delta t}{\rho}(\partial_{x,h}\sigma_{xy}^n + \partial_{y,h}\sigma_{yy}^n + \sum_{l=1}^{N_{\text{SLS}}}(\partial_{y,h}R_{1,l}^n - \partial_{y,h}R_{xx,l}^n + \partial_{x,h}R_{xy,l}^n)) \quad (5.110)$$

$$\sigma_{xx}^{n+1} =\sigma_{xx}^n + \Delta t\left((\lambda+2\mu)\partial_{x,h}v_x^{n+1/2} + \lambda\partial_{y,h}v_y^{n+1/2}\right) \quad (5.111)$$

$$\sigma_{yy}^{n+1} =\sigma_{yy}^n + \Delta t\left(\lambda\partial_{x,h}v_x^{n+1/2} + (\lambda+2\mu)\partial_{y,h}v_y^{n+1/2}\right) \quad (5.112)$$

$$\sigma_{xy}^{n+1} =\sigma_{xy}^n + \Delta t\mu\left(\partial_{x,h}v_y^{n+1/2} + \partial_{y,h}v_x^{n+1/2}\right) \quad (5.113)$$

$$R_{1,l}^{n+1} =\frac{1}{1-0.5\Delta t\theta_{1,l}}\left(R_{1,l}^n + \frac{\Delta t}{2}\theta_{1,l}R_{1,l}^n + 0.25\Delta t\frac{\theta_{1,l}D_{1,l}}{Q_1+\sum_{l=1}^{N_{\text{SLS}}}D_{1,l}}(\sigma_{xx}^{n+1} + \sigma_{yy}^{n+1} + \sigma_{xx}^n + \sigma_{yy}^n)\right) \quad (5.114)$$

$$R_{xx,l}^{n+1} =\frac{1}{1-0.5\Delta t\theta_{2,l}}\left(R_{xx,l}^n + \frac{\Delta t}{2}\theta_{2,l}R_{xx,l}^n + 0.25\Delta t\frac{\theta_{2,l}D_{2,l}}{Q_2+\sum_{l=1}^{N_{\text{SLS}}}D_{2,l}}(\sigma_{xx}^{n+1} - \sigma_{yy}^{n+1} + \sigma_{xx}^n - \sigma_{yy}^n)\right) \quad (5.115)$$

$$R_{xy,l}^{n+1} =\frac{1}{1-0.5\Delta t\theta_{2,l}}\left(R_{xy,l}^n + \frac{\Delta t}{2}\theta_{2,l}R_{xy,l}^n + 0.5\Delta t\frac{\theta_{2,l}D_{2,l}}{Q_2+\sum_{l=1}^{N_{\text{SLS}}}D_{2,l}}(\sigma_{xx}^{n+1} + \sigma_{xy}^n)\right) \quad (5.116)$$

CPML boundary condition As we described in section 5.3.5 are all the boundaries of the grid artificial and outgoing waves should be absorbed there in order to simulate a semi-infinite medium (see figure 5.15). We use the unsplit CPML technique of [83], also analyzed by [88], which consists of modifying each spatial derivative along the direction perpendicular to the absorbing layer. We describe it exemplarily for the x component, and the spatial derivative is replaced by

$$\partial_{\tilde{x}} = \frac{1}{\kappa_x}\partial_x + \psi_x, \quad (5.117)$$

where ψ_x is a memory variable whose time evolution is governed at each time step by an additional equation:

$$\psi_x^n = b_x\psi_x^{n-1} + a_x\left(\partial_x\right)^{n-\frac{1}{2}} . \quad (5.118)$$

This implies that significantly more equations need to be solved in the PML regions, in particular near the corners of the grid, because contributions coming from the PML layers located along x and y are summed there and one memory variable and thus a time evolution equation is needed for each, however this is acceptable because the PML regions are small compared to the rest of the model.

The coefficients a_x and b_x in the PML, which do not vary with time, are given by:

$$b_x = e^{-(d_x/\kappa_x+\alpha_x)\Delta t} \tag{5.119}$$

and

$$a_x = \frac{d_x}{\kappa_x(d_x+\kappa_x\alpha_x)}(b_x-1), \tag{5.120}$$

where $\kappa_x \geq 1$, $d_x \geq 0$ and $\alpha_x \geq 0$ are three real damping coefficients. We refer the reader to [83] for more details.

Gradient of the objective function The gradient of the objective function results as integral over time in which both the solution of the forward problem and the adjoint problem occurs. The adjoint problem runs backward in time, which means that when the Fréchet derivatives are calculated, the i-th time step of the adjoint problem is combined with the $(N-i)$-th time step of the forward problem. However, this would mean that the entire forward modeling would have to be stored. Since real-world problems are so large that this is practically impossible, there are several ways to store the corresponding time steps of the forward and adjoint problems at the same time without storing the entire forward run. These are shown in figure 5.18, where (a) is the described standard variant

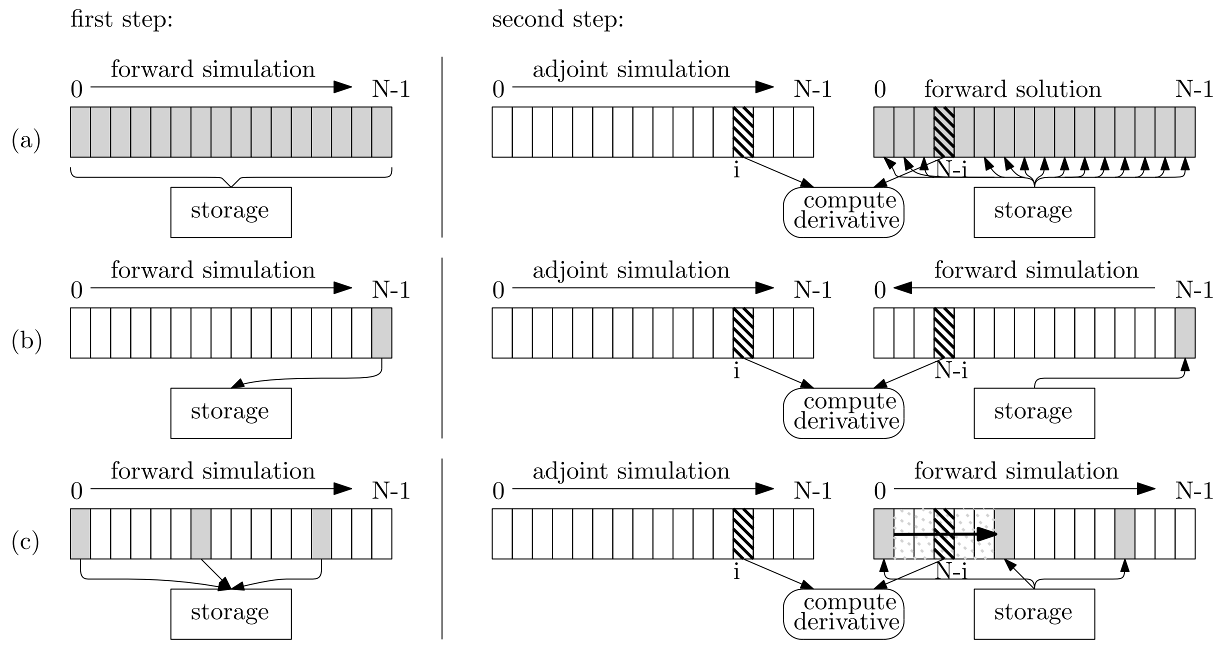


Figure 5.18: Three different strategies to store the solution of the forward simulation (inspired by Komamtitsch et al. [84]).

where the entire forward simulation is stored and then loaded from memory during the adjoint simulation to calculate the derivative. A very memory efficient variant is variant (b) where only the last time step of the forward simulation is stored. Afterwards, another forward simulation is run parallel to the adjoint simulation but this time backward in time. However, this variant cannot be applied (without strong filtering and the resulting significant loss of accuracy (e.g., Ammariet et al. [11]) in the anelastic case or in the presence of any kind of energy loss, because the temporal energy decay is unstable from a numerical point of view (e.g., Liu and Tromp [96, 97]). The reason for this is that in adjoint simulation the fields are amplified to restore the original energy, but this also amplifies the (numerical) noise and thus the methods become unstable. Thus, this variant is not applicable in our case. Therefore we use the third variant (c) which is stable even when energy is dissipated. In this approach, in the first stage we store a small number of equally spaced checkpoint/restart 'files' of the forward modeling, typically every few hundred or thousand time steps and in the second stage we still run two simulations simultaneously, an adjoint run and a forward run, but instead of running the forward run backward in time from the stored last time step, we run it in blocks in reverse order, but in forward direction within each block. In any case, we start from the previous restart file and store only this part of the run in memory. Since the run is performed forward in time and not backward, this process is always stable, even in the presence of damping.

5.5.3 Full waveform inversion algorithm

Previously we have only considered one source due to simplification. In practice, this is of course not feasible to achieve sufficiently good quality, therefore up to several 100 sources are quite common. Since each source represents an independent simulation, we call this an event, i.e., in practice, the different sources are executed with a time offset. This gives us another loop in our FWI algorithm and the result is algorithm 5.2.

Algorithm 5.2 (simplified FWI algorithm).

```
1: **Input:** initial model $m_0$, source signals $\mathbf{s}$, receiver signals $\mathbf{r}$
2: **Output:** model $m$
3: $i = 1$
4: **while** $|m_i - m_{i-1}| < TOL$ **do**
5:     **for** $(j = 0; j < \#events; j++)$ **do**
6:         forward simulation for event $j$                          ▷ eqs. (5.101)–(5.108)
7:         adjoint simulation for event $j$                          ▷ eqs. (5.109)–(5.116)
8:         calculate derivative of misfit function for event $j$     ▷ see section 5.4.5
9:         add derivative for event $j$ to full derivative
10:    **end for**
11:    update of optimization method($m_{i+1}$, $m_i$)
12:    $i = i + 1$
13: **end while**
```

Parallelization A very trivial parallelization is the parallel execution of different events. Because they represent independent simulations, parallelization is possible without further modifications. But the degree of parallelism is automatically limited. Another possibility is to parallelize the single simulations of the single events, i.e., to parallelize the forward and adjoint simulation. For this purpose, we choose a domain decomposition approach in which the domain is divided into overlapping subdomains. The overlapping ensures convergence and consistency [50]. Figure 5.19 shows the overlapping domain decomposition schematically.

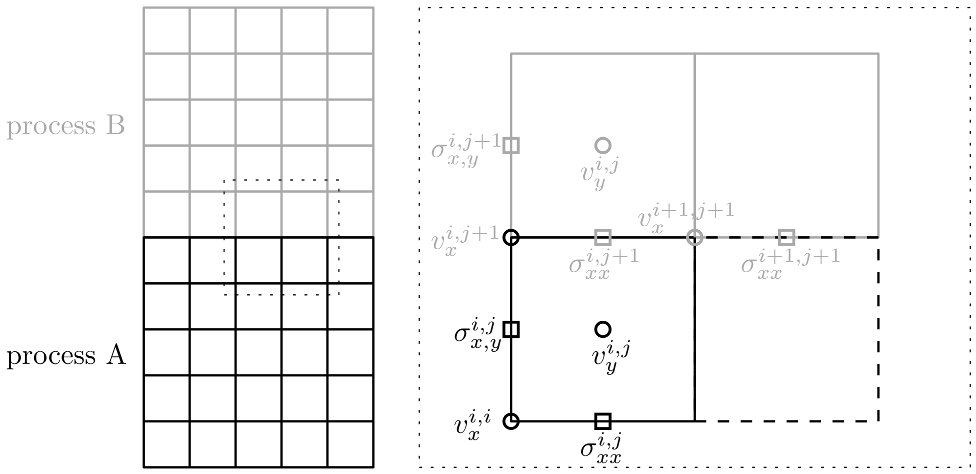


Figure 5.19: Overlapping domain decomposition for two processes. On the right the degrees of freedom for the black cell are shown. All grey degrees of freedom, which actually belong to process B, represent so-called halo degrees of freedom for this cell.

On the left side, the division into two subdomains for process A and process B is shown. On the right side of figure 5.19 we look at a cell of process A that is adjacent to process B. In black are all parameters, which are degrees of freedom of this cell. For the update calculation of these degrees of freedom, information is needed that does not actually belong to process A, these are called halo degrees of freedom. They also exist on process A, but are not updated there, this happens on process B. After each time step, the current values of these halo degrees of freedom are transferred from process B to process A. If a higher discretization method is used than the central difference quotient, the number of halo degrees of freedom increases. For more details on domain decomposition procedures, we refer to [50].

GPU acceleration GPUs are very well suited as a target platform because we use the finite difference method and the Crank-Nicolson method for discretization, and these are very structured methods. An efficient implementation of such stencil based methods is already well researched and we refer to chapter 2 for general information or the literature [43, 109, 135] for further information.

Multiscale seismic waveform inversion Being a local inversion method, FWI carries the danger of ending up in one of the possibly numerous local minima of the objective function. It is possible that the final model explains the observed data well, even if it is not the correct model. In other words, the data misfit is small, but the model misfit is high. A better initial model may be required to alleviate this problem. Different strategies of inversion, like the multilevel inversion, can help to avoid stepping in local minima and thus are needed for a successful inversion. Using the so-called multilevel approach, we do

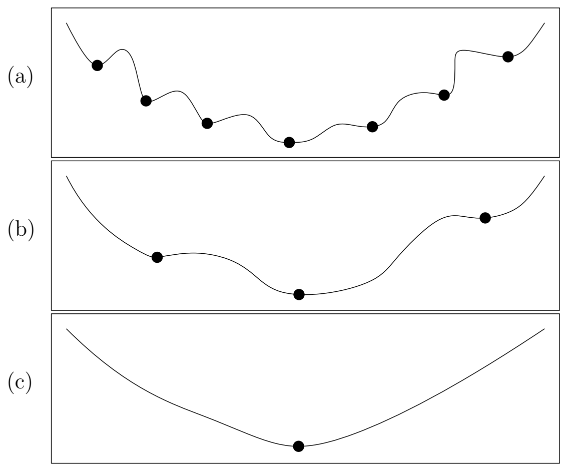


Figure 5.20: Misfit function for different frequency bands (from high frequencies at the top, to low frequencies at the bottom) (inspired by Bunks et al. [39]).

not invert all data at the same time, but separate the inversion into levels, where the data is limited/filtered to a certain frequency band. This can help to reduce the non-linearity of inversion and prevent being trapped in local minima. We start from low frequencies and increase the frequencies during inversion [39]. More precisely, inversions are performed on the different frequency levels one after the other. First, an inversion is performed on the lowest level. Then, the resulting model is used as the initial model for the inversion at the next level, and so on. Especially for more complex problems, this strategy is essential to reach the global misfit minimum. The main justification is, that the form of the misfit function is strongly influenced by the frequency range of the data. Figure 5.20 illustrates this, where the misfit function is plotted for different scale lengths that increase from (a) to (c). As the scale length increases and thus the frequency decreases, the misfit function becomes smoother and the number of local minima decreases. Therefore, to reach the global minimum, a model for low frequencies can be much further away from the real model, while a much closer model is required for high frequencies. In particular, this means that details such as thin regions do not appear in the resulting model, when inverting on the low frequencies scale.

We thus add another loop over the different frequency bands to be inverted and obtain our final FWI algorithm, which is shown in algorithm 5.3.

Algorithm 5.3 (FWI algorithm).

```
1: Input: initial model $m_0$, source signals $\mathbf{s}$, receiver signals $\mathbf{r}$
2: Output: model $m$
3: for frq= 0;frq<#frequencies;frq++) do
4:     $i = 1$
5:     while $|m_i - m_{i-1}| < TOL$ do
6:         for ($j = 0; j <$#events$; j++$) do
7:             forward simulation for event $j$                        ▷ eqs. (5.101)–(5.108)
8:             adjoint simulation for event $j$                        ▷ eqs. (5.109)–(5.116)
9:             calculate derivative of misfit function for event $j$   ▷ see section 5.4.5
10:            add derivative for event $j$ to full derivative
11:        end for
12:        update of optimization method($m_{i+1}$, $m_i$)
13:        $i = i + 1$
14:    end while
15: end for
```

5.5.4 Computation of Zener model parameters

In order to determine the relaxation parameters $d_{1,l}$, $\theta_{1,l}$ and $d_{2,l}$, $\theta_{2,l}$ for $l = 1, \ldots, N_{\mathrm{SLS}}$ for the generalized Zener Model, the minimization problem (5.67) must be solved. In principle, any quadratic optimization problem solver can be used. We use the SolvOpt algorithm, which was developed by Franz Kappel and Alexei V. Kuntsevic [89], since this algorithm yields good results for this application as shown in [26].

5.6 Numerical results

In this section, we evaluate the impact of viscosity on the inversion process and whether it is necessary to take viscosity into account in the inversion process. Furthermore, we compare different regularization methods and present a methodology that yields better inversion results and converges faster. For this purpose, we first use a simple model problem and then apply our findings to the established Marmousi example. Finally, we evaluate the scalability of our implementation.

5.6.1 Simple test case

The challenging aspect of geophysical models is that the domains usually consist of subdomains with constant material properties. There is often no smooth transition between two layers, but areas with different physical parameters are directly adjacent to each other.

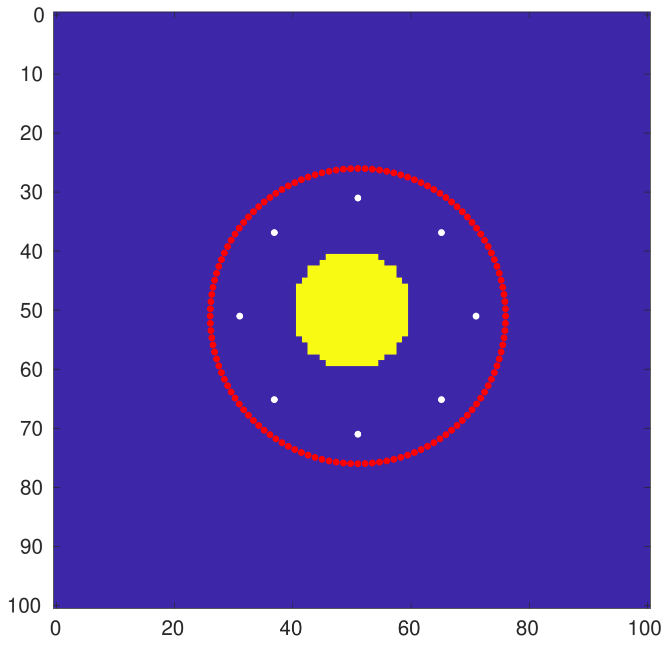


Figure 5.21: Simple example with two different regions (blue and yellow). The white points describe the positions of the sources, the red points describe the locations of the receivers.

A simple model consisting of two different areas is the bubble example shown in figure 5.21. It is a square of size $100m \times 100m$, with a circular region with other physical properties in the center. That region has a radius of $10m$. We consider here two different versions, one with a higher attenuation and one with a lower attenuation. The corresponding values for the densities ρ, the velocities of the p- and s-waves, and the Q factors for the two different versions are given in table 5.1. This or similar simple examples are often used in geophysics to validate regularization techniques or other methods [4, 24, 66].

Test case	Region	ρ in kg/m^3	v_p in m/s	v_s in m/s	Q_1	Q_2
high attenuation	inner	1100	1600	1000	100	85
	outer	1000	1500	900	65	55
low attenuation	inner	1100	1600	1000	220	185
	outer	1000	1500	900	100	85

Table 5.1: Physical properties of the inner and outer region (yellow and blue in figure 5.21) for a low and high attenuation example.

We use 8 sources distributed equidistantly radially around the bubble region (with a distance of $20m$ from the center of the region). The sources are shown as white dots in figure 5.21. Here, the sources are shot sources, i.e., pressure is given over time. The number of receivers is 128 and they record velocity values. They are also distributed equidistantly radially with a distance of $25m$ from the center of the area. The receivers are shown as red dots in figure 5.21. As a source time function we use the shifted Ricker wavelet [134], which is defined as follows

$$s(t) = A\left(1 - \frac{1}{2}\omega_0^2 t^2\right)\exp\left(-\frac{1}{4}\omega_0^2 + t^2\right) - t_0$$

where t is the time, t_0 is the time shift, A is the amplitude and ω_0 is the most energetic frequency (in radians per second). We choose a dominant frequency of f_0=100, with $\omega_0 = 2\pi f_0$, a time shift $t_0 = 0.025$ and an amplitude $A = 10^7$. For the modified Zener model, we use $N_{\text{SLS}} = 3$ parallel Zener models, where we consider the objective function (5.67) in the frequency domain of $[\exp(\log(f_0) - \log(12)\cdot 0.5),\ 12\cdot\exp(\log(f_0) - \log(12)\cdot 0.5)] \approx [28.87,\ 346.41]$ with $K = 12$ discrete frequencies. For the high attenuation case, we set the Q range of both Q factors to $[55,\ 100]$. For the low attenuation case we set $Q_1 \in [90,\ 230]$ and $Q_2 \in [75,\ 200]$.

As initial model we always use a homogeneous region with the physical properties of the outer region and we generate objective data by running a forward simulation with the true model.

We use l-BFGS to solve the optimization problem, where we set $l = 20$ (so the gradient is calculated based on the last 20 gradients) and continue solving until no better solution can be found (cf. conditions from section 5.5.1 on page 140). In cases where the split-Bregman solver is used, the iteration count for this solver is set to 10.

Here, except in section 5.6.5, we always invert in the entire frequency space, in other words, we do not use a multiscale approach.

5.6.2 Impact of the viscosity

The effort for the viscoelastic inversion is much higher than for the pure elastic inversion. This is especially due to the fact that $3 \times N_{\text{SLS}}$ partial equations have to be solved additionally. Furthermore, the memory overhead is higher, as well as the implementation overhead, since the forward simulation does not correspond to the adjoint simulation as it is in the elastic case. Therefore, we want to use the bubble example to investigate whether the more expensive viscoelastic inversion is really necessary and how big the difference in accuracy is.

For this purpose, we consider both, the example with the higher attenuation and the one with the lower attenuation. As objective data we always use the data resulting from the forward simulation of the viscoelastic equation with the true model. For the inversion we use different approaches: The first approach is to simply omit the viscosity completely and use the pure elastic equations in both, the forward and adjoint simulation. The second approach is using the viscoelastic equations for the forward simulation and the purely elastic PDEs in the adjoint simulation. The third approach is to use the viscoelastic equations in the forward simulation as well as in the adjoint simulation, whereby we again distinguish whether we optimize all model parameters $m = (\rho,\ v_p,\ v_s,\ Q_1,\ Q_2)$ or only $\rho,\ v_p$ and v_s.

The resulting data misfits of the objective function for all variants and both test cases are shown in table 5.2, where we have normalized the results by the result of the second variant. Using the elastic equations for both the forward and the adjoint simulation, the data misfit is 52 times larger for the high attenuation and 19 times larger for the case with less attenuation. If we also use the viscoelastic equations in the adjoint simulation and invert all parameters, the data misfit decreases by a factor of 47 for the high attenuation and by a factor of 5 for the lower attenuation. However, using the viscoelastic PDEs in both simulations but inverting only the model parameters ρ, v_p and v_s leads to only a small reduction of the data misfit. Overall, the differences are larger for the high attenuation

	both elastic	elastic adjoint	viscoelastic inv. for ρ, v_p, v_s	viscoelastic inv. for all parameters
high attenuation	52.727	1.000	0.950	0.021
low attenuation	19.169	1.000	0.998	0.197

Table 5.2: Data misfit for the low and high attenuation example using elastic or viscoelastic equations for forward and adjoint modeling in the inversion algorithm.

case than for the low attenuation case. This corresponds exactly to the expectation that the influence of viscosity is larger for high attenuation.

The same behavior can be seen when considering the model misfit, which is shown in table 5.3. Here, the model misfit even increases at the step from elastic to viscoelastic adjoint simulation, if only ρ, v_p and v_s are optimized. The reason for this can be explained by figure 5.22, which shows the profiles of the resulting model parameters along $x = 50m$.

If we consider figure 5.22c we see that the velocity in the inner region is faster than the velocity in the true model. This is because the Q factor in this region is smaller than in the true model, which makes the attenuation larger. The higher velocities try to counteract this effect. This effect is slightly higher in the case where both the forward and the adjoint simulation are realized with the viscoelastic equations, since the effect occurs in both simulations.

	both elastic	elastic adjoint	viscoelastic inv. for ρ, v_p, v_s	viscoelastic inv. for all parameters
high attenuation	13.294	1.000	1.057	0.397
low attenuation	1.874	1.000	1.005	0.555

Table 5.3: Model misfit for the low and high attenuation example using elastic or viscoelastic equations for forward and adjoint modeling in the inversion algorithm.

Findings and implications for the following sections For simulations with low attenuation and good initial model of the Q factors, the use of elastic adjoint simulation is sufficient, since the error is small and does not affect the inversion significantly.

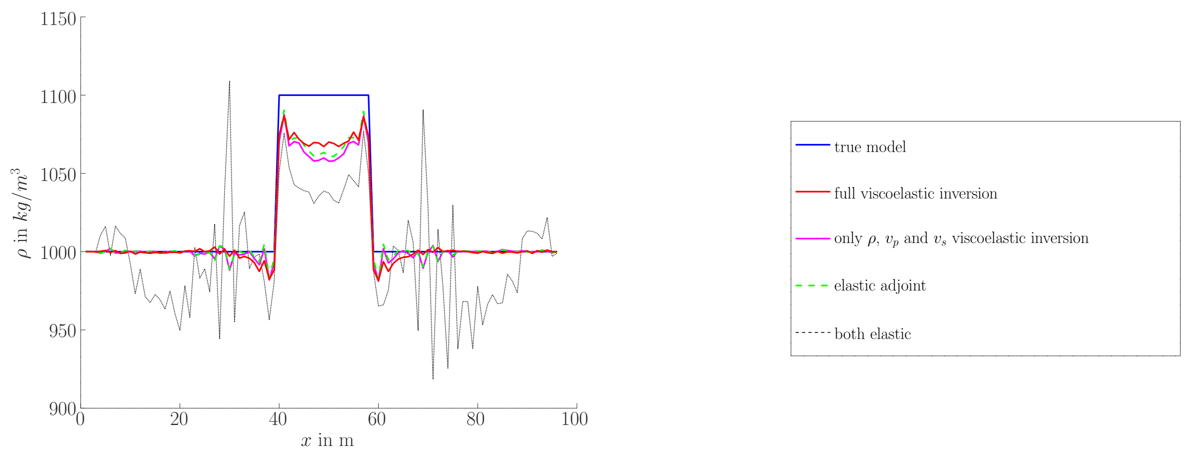
If high attenuation occurs in a simulation, the viscoelastic PDE must also be used in the adjoint simulation, otherwise the error is too large and the inversion of the p- and s-wave velocities is also faulty. In addition, all model parameters should be inverted in any case, otherwise the use of the viscoelastic adjoint simulation does not provide any advantage over the elastic simulation.

Therefore, for the investigations in the following sections, we always use the viscoelastic wave equation in the forward and adjoint simulation and invert all model parameters.

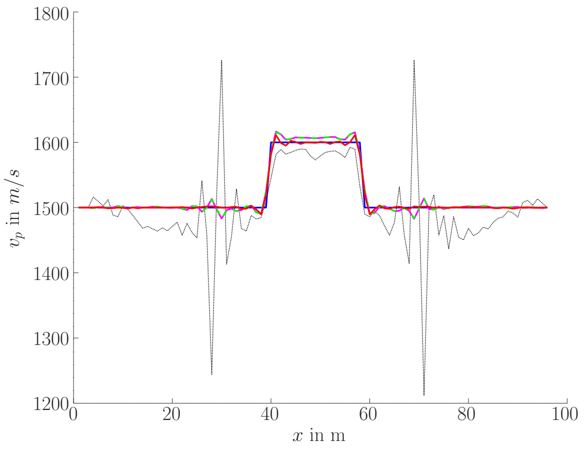
5.6.3 Comparison of different regularizations

In this section we compare the different regularization methods from section 5.4.1. For this purpose, we consider, based on the findings from the last section, the inversion of all model parameters of the viscoelastic wave equation. The aim of a regularization is to improve the robustness of the ill-posed inverse method against perturbed initial data. Similarly, regularization can lead to faster convergence. Here we consider the higher attenuation example and consider both unperturbed and white noise perturbed input data.

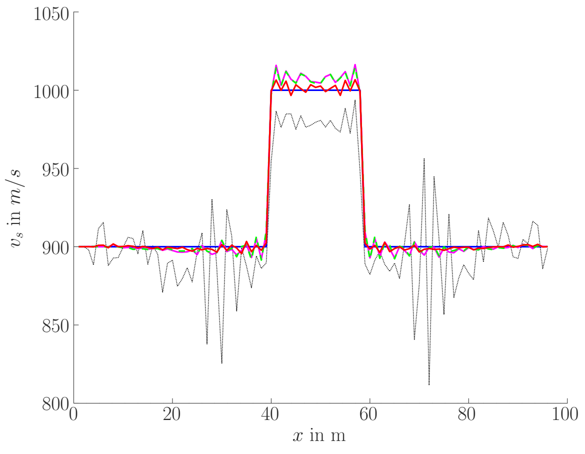
To determine a good regularization weight for the different procedures (see section 5.4.1), we test the following weights $\lambda_{\mathcal{R}_0}, \lambda_{\mathcal{R}_1}, \lambda_{\mathcal{R}_2} \in \{10^{-20},\ 7.5\cdot10^{-21},\ 5\cdot10^{-21},\ 2.5\cdot10^{-21},\ 10^{-21},\ \dots,\ 10^{-30}\}$ for the Tikhonov regularization methods. The regularization weights are of the order 1E-20– 1E-30 here, because the misfit of the unregularized optimization problem (5.80) is small, of the order 1E-10– 1E-12, and the ℓ_2 norm of the model parameter (or the norm of the gradient, Laplacian of the model parameter) can become very large. The resulting data and model misfits for the unperturbed initial data can be found in tables A.1–A.2 and for the perturbed ones in tables A.6–A.7 in section A.1.7 in the appendix.



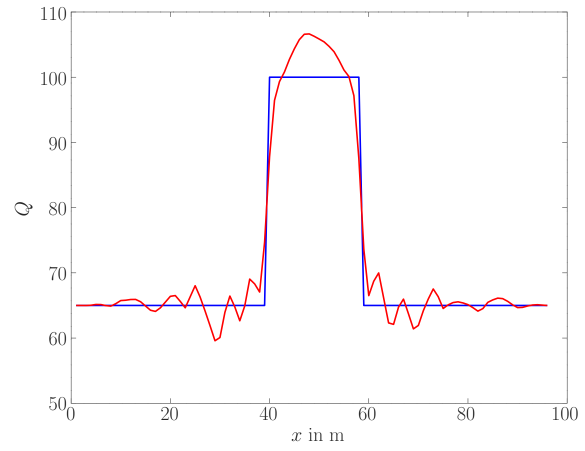
(a) Density ρ



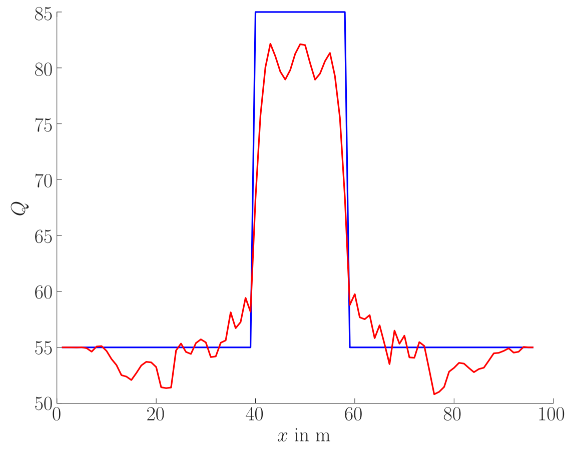
(b) p-wave velocity v_p



(c) s-wave velocity v_s



(d) Quality factor Q_1



(e) Quality factor Q_2

Figure 5.22: Profiles of the simple example along $x = 50m$ for different inversion strategies. We omit the 'on only ρ, v_p and v_s gradient', 'elastic adjoint' and 'both elastic' variants in figures 5.22d and 5.22d, since they are not part of the inverse and due to this constant 65 and 55 as the initial model.

As recommended in the literature [58, 94] we set the regularization weight for the total variation and the generalized p-variation regularization as

$$\lambda_{\mathrm{TV},1} = \kappa_{\mathrm{TV}} \frac{\|\nabla\chi\|_2^2}{\|m - v + q\|_2^2}, \tag{5.121}$$

$$\lambda_{\mathcal{T}_p,1} = \kappa_{\mathcal{T}_p} \frac{\|\nabla\chi\|_2^2}{\|m - v + q\|_2^2}, \tag{5.122}$$

where v and q are auxiliary and ADMM variables, since we use an alternating approach to solve the optimization problem (see section 5.5.1 on page 141).κ_{TV}. $\kappa_{\mathcal{T}_p}$ are scalar regularization weights to adjust the effect of the regularization depending on the problem. We point out that this choice of the regularization weight depends on the model parameter and thus must also be taken into account when determining the gradient of the objective function. Analogous to [58, 94], we consider the regularization parameter in the determination of the gradient as a constant factor. To determine a good regularization weight for the total variation and the generalized p-variation regularization, we test the following weights $\kappa_{\mathrm{TV}}, \kappa_{\mathcal{T}_p} \in \{10^{-1},\ 7.5 \cdot 10^{-2},\ 5 \cdot 10^{-2},\ 2.5 \cdot 10^{-2},\ 10^{-2}, \dots,\ 10^{-10}\}$.

The implementation of the TV and p-variation regularization using the split-Bregman method requires the setting of additional parameters (see section A.1.6 in the appendix). For TV regularization, we choose $\lambda_{\mathrm{TV},2} = 0.7$ and $\alpha = 2\lambda_{\mathrm{TV},2}$ as parameters in the split-Bregman method, as suggested in the literature [94] . For the p-variation regularization, we choose similar to [58] $\mu = 0.5$, $p = 0.5$, $\alpha_0 = 2$, $\alpha_1 = 1$, $\eta_= \zeta \cdot \mu$, and $\eta_1 = \alpha_1/\alpha_0 \cdot \eta_0$, where we can use ζ to regularize strongness and set $\zeta = 2$ by default. This means that the larger ζ is chosen, the stronger the weighting of the p norm.

Unperturbed objective data We first consider the simulation for unperturbed data and compare the data and model misfit of the last iteration prior to convergence detection. Since the solver terminates prematurely when no improved solution is obtained with respect to the criteria in section 5.5.1, different numbers of iterations are performed depending on the regularization method. In the left half of table 5.4 the best results regarding data and model misfit are listed for the different regularization methods and for using no regularization, where the results were normalized with the initial misfit. Here 'N/A' indicates that no improvement was achieved compared to the non-regularized solution. In this case, we obtain a small improvement in the data misfit only for $\mathcal{R}_1$.

However, if we look at the 100th iteration when solving with the different regularization methods, all regularization methods provide advantages with respect to data and model misfit. The TV regularization performs best here, which can achieve the best results for both the data and the model misfit, as shown in the right-hand side of table 5.4. For the unperturbed data, regularization can thus lead to faster convergence but not to better results. This should not be surprising, however, since regularization implements another constraint that the exact solution does not necessarily have to satisfy (everywhere). It should be noted that regularization can lead to better results in a fixed time budget, however, this is not examined further here.

	Last iteration				Iteration 100			
	data misfit		model misfit		data misfit		model misfit	
	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit
no		1.39E-5		4.64E-2		1.74E-3		1.24E-1
$\mathcal{R}_0$	N/A	N/A	N/A	N/A	2.5E-22	7.65E-4	1.0E-26	8.88E-2
$\mathcal{R}_1$	5.0E-25	1.31E-5	N/A	N/A	5.0E-21	9.54E-4	2.5E-25	9.10E-2
$\mathcal{R}_2$	N/A	N/A	N/A	N/A	5.0E-23	6.78E-4	7.5E-28	9.23E-2
$\mathcal{R}_{\mathcal{T}_p}$	N/A	N/A	N/A	N/A	2.5E-03	6.91E-4	1.0E-04	8.84E-2
$\mathcal{R}_{\text{TV}}$	N/A	N/A	N/A	N/A	2.5E-03	6.02E-4	7.5E-08	8.63E-2

Table 5.4: Best model and data misfit for the simple example achieved by different regularization methods for unperturbed data. 'N/A' indicates that no improvement was achieved compared to the non-regularized solution. $\mathcal{R}_*$ denotes the regularization method and $\lambda_{\mathcal{R}}$ the corresponding regularization weight.

Perturbed objective data Since regularization methods should ensure robustness of the method against perturbed initial data, we consider perturbed initial data in the following. For this purpose, we add a normally distributed Gaussian noise with a variance of 10^{-12} to the initial data obtained by performing the forward simulation for the true model. This results in a signal to noise ratio (SNR) SNR_{DB} between -0.22 and 28.12 dB. The SNR is defined as follows:

$$\text{SNR} = \left(\frac{\text{RMS}(s)}{\text{RMS}(s)}\right)^2$$
$$\text{SNR}_{\text{DB}} = 10\log_{10}(\text{SNR})$$

Here RMS is the root mean square amplitude:

$$\text{RMS} = \sqrt{\frac{1}{N_{\text{sampl.}}}\sum_{i=1}^{N_{\text{sampl.}}} s_i^2}$$

Here s describes the signal or the received data at one receiver, s_i the i-th sample value of the signal or in our case the i-th entry of the objective data and $N_{\text{sampl.}}$ the number of samples, in our case the number of received data at one receiver.

In this case, all regularization methods can lead to improvements, as we can see in table 5.5, where the best possible results with regard to data and model misfit are listed for the different regularization methods as well as the corresponding regularization weight. The data misfit differs barely when using the different regularization methods and is with 0.3282 slightly better than without regularization with 0.3287. The best model misfit is achieved by the p-variation regularization method with 0.1129 compared to 0.1166 without regularization. However, if we compare the results of the unperturbed data with the results of the perturbed data, we see that the data and model misfit can be reduced less when using perturbed data than using unperturbed data. The data misfit becomes larger for disturbed input data, since the noise is also transmitted through the simulation and thus arrives at the receivers. That the model misfit increases has to do in particularly with the ill-posedness of the inverse problem.

	Last iteration				Iteration 100			
	data misfit		model misfit		data misfit		model misfit	
	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit
no		0.3287		0.1166		0.3287		0.1166
$\mathcal{R}_0$	5.0E-23	0.3282	7.5E-24	0.1134	2.5E-28	0.3284	1.0E-23	0.1126
$\mathcal{R}_1$	1.0E-21	0.3282	2.5E-26	0.1130	5.0E-22	0.3284	1.0E-26	0.1126
$\mathcal{R}_2$	1.0E-25	0.3282	1.0E-24	0.1132	7.5E-27	0.3284	1.0E-25	0.1126
$\mathcal{R}_{\mathcal{T}_p}$	1.0E-10	0.3282	2.5E-06	0.1129	1.0E-08	0.3284	1.0E-10	0.1126
$\mathcal{R}_{\text{TV}}$	5.0E-10	0.3282	2.5E-03	0.1138	5.0E-04	0.3284	1.0E-05	0.1133

Table 5.5: Best model and data misfit for the simple example achieved by different regularization methods for perturbed data. $\mathcal{R}_*$ denotes the regularization method and $\lambda_{\mathcal{R}}$ the corresponding regularization weight.

Findings and implications for the following sections In the simulation of undisturbed objective data, no regularization is necessary. However, the use of regularization can accelerate convergence. In case of noisy objective data, all presented regularization methods lead to improvements. The p-variation regularization method provides the best results.

The achieved model misfit for inverting noisy data is much higher than for unperturbed data. Therefore we present in the following a method, with which the model misfit can be better minimized also for disturbed data.

5.6.4 p-variation reduced gradient inversion

In the last subsection we have seen that the presented regularization methods lead to an improvement of the inversion for disturbed data, but the model can be reconstructed worse than for unperturbed data. The best results were obtained with the p-variation method, so that we further modify this method by using the principle of this method even stronger. For p-variation regularization we use, so far, an alternating method as outlined in section 5.5.1 on page 141. Roughly speaking, first an update is made, which ignores the total p-variation regularization, and then the new model is improved with respect to the total generalized p-variation.

The idea is to directly improve the gradient of the objective function (5.80) with respect to the p-norm and to use the improved gradient to perform the update in the l-

BFGS algorithm, so that the second step can be omitted. More precisely, we first compute the gradient of the objective function

$$g = \nabla \chi(m)$$

as described in section 5.4.5. Next, we search a modified gradient $g_{\mathcal{T}_p}$, which has the smallest $\mathcal{T}_p$ distance to g, i.e.,

$$g_{\mathcal{T}_p} = \min_{\bar{g}} \left\{ \frac{1}{2} \|g - \bar{g}\|_2^2 + \lambda_{\text{p-vrg}} \mathcal{T}_p(\bar{g}) \right\},$$

where $\lambda_{\text{p-vrg}}$ controls how strong the reduction of the p-variation is. Then, we use $g_{\mathcal{T}_p}$ instead of g in the l-BFGS algorithm. We call this modification p-variation reduced gradient inversion. For solving we use the split-Bregman algorithm (cf. section A.1.6) and use the same parameters as for the p-variation regularization.

Since this change of the gradient may smooth out information that does not represent oscillations but enhances narrow regions, we use this strategy only as a presolver. We recall that the p-variation reduced gradient method reduces strong variations, since this is a characteristic of oscillations. However, this also applies to thin regions that are surrounded by regions that have parameters that deviate strongly from their own physical parameters, but the neighboring regions have similar physical properties. This means that after solving with the p-variation reduced gradient inversion method, a further solving with the standard, possibly regularized with an arbitrary method, inversion is performed. We additionally use the p-variation regularization in the main solver to further consider the reduced p-variation norm of the model.

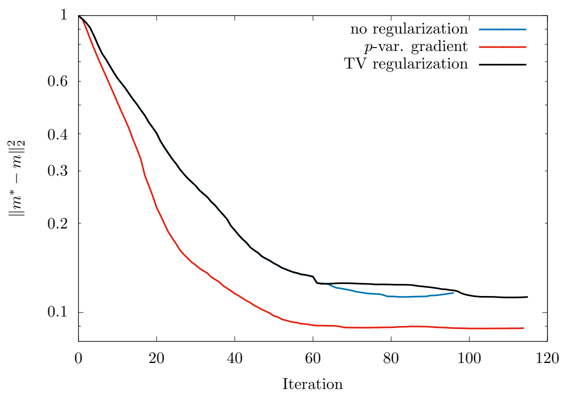
In the following we investigate the influence of the p-variation reduced gradient inversion for the simulation with perturbed and unperturbed objective data.

Perturbed objective data We apply the modified p-variation reduced gradient inversion method to the bubble example with noisy objective data, and set $\zeta = 1.15$ and $\kappa_{\mathcal{T}_p} = 1\text{E-}7$. The results and the best results from the previous section are shown in table 5.6. In terms of the resulting data misfit, this method does not achieve any significant improvement, but the model misfit is reduced by 24 % compared to no regularization and by 21 % compared to TV regularization. Furthermore, the error reduction for both the model and data misfit is faster when p-variation reduced gradient inversion is used, as shown in figure 5.23.

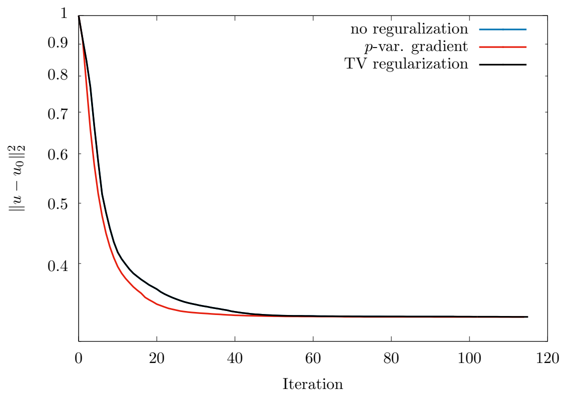
If we consider the profiles of the results along $x = 50m$, which are shown in figure 5.24, we see that the results for the p-variation reduced gradient inversion show less noise/oscillations outside the inner region. Furthermore, the Q factors Q_1 and Q_2 are better inverted overall.

	Last iteration				Iteration 100			
	data misfit		model misfit		data misfit		model misfit	
	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit
no		0.3287		0.1166		0.3287		0.1166
$\mathcal{R}_{\mathcal{T}_p}$	1.0E-10	0.3282	2.5E-06	0.1129	1.0E-08	0.3284	1.0E-10	0.1126
$\mathcal{R}_{\text{TV}}$	5.0E-10	0.3282	2.5E-03	0.1138	5.0E-04	0.3284	1.0E-05	0.1133
p-vrg	1E-7	0.3283	1E-7	0.0887	1E-7	0.3283	1E-7	0.0884

Table 5.6: Best model and data misfit for the simple example achieved by the new p-variation reduced gradient inversion compared to the best regularization methods from table 5.5. $\mathcal{R}_*$ denotes the regularization method and $\lambda_{\mathcal{R}}$ the corresponding regularization weight. The third line 'p-vrg' are the results of the p-variation reduced gradient inversion.



(a) model misfit



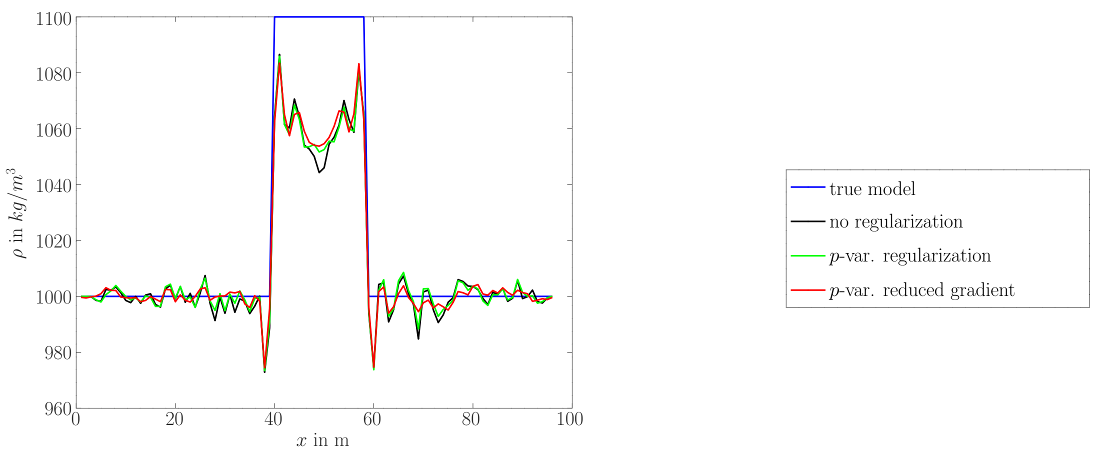
(b) data misfit

Figure 5.23: Convergence of the data and model misfit for the simple example using perturbed objective data.

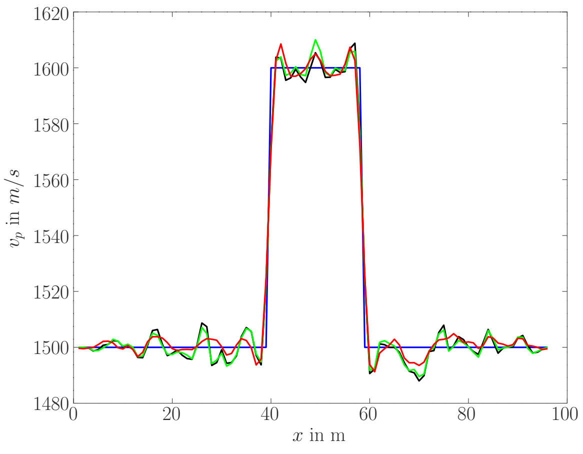
Unperturbed objective data Next, we apply the new p-variation reduced gradient inversion method to the bubble example with unperturbed objective data, and set $\zeta = 1.05$ and $\kappa_{\mathcal{T}_p} = 1\text{E-}9$. We recall that none of the previously considered regularization methods provided better results in terms of model and data misfit than using no regularization. The results of the p-variation reduced gradient inversion and the best results from the previous section, see table 5.4, are shown in table 5.7. The p-variation reduced gradient

	Last iteration				Iteration 100			
	data misfit		model misfit		data misfit		model misfit	
	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit	$\lambda_{\mathcal{R}}$	misfit
no		1.39E-5		4.64E-2		1.74E-3		1.24E-1
$\mathcal{R}_1$	5.0E-25	1.31E-5	N/A	N/A	5.0E-21	9.54E-4	2.5E-25	9.10E-2
$\mathcal{R}_{\text{TV}}$	N/A	N/A	N/A	N/A	2.5E-03	6.02E-4	7.5E-08	8.63E-2
p-vrg	1.E-10	1.07E-5	1.E-10	4.02E-2	1.E-10	5.73E-4	1.E-10	7.92E-2

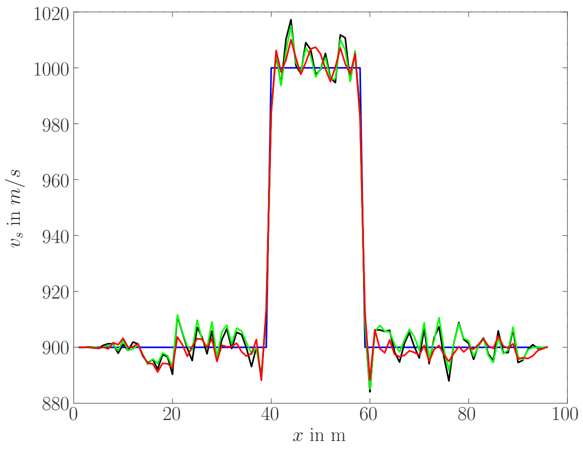
Table 5.7: Best model and data misfit for the simple example achieved by the new p-variation reduced gradient inversion compared to the best regularization methods from table 5.4. $\mathcal{R}_*$ denotes the regularization method and $\lambda_{\mathcal{R}}$ the corresponding regularization weight. The fourth line 'p-vrg' are the results of the p-variation reduced gradient inversion.



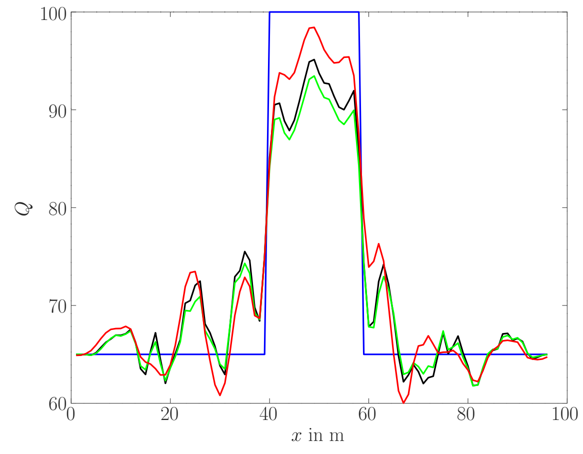
(a) Density ρ



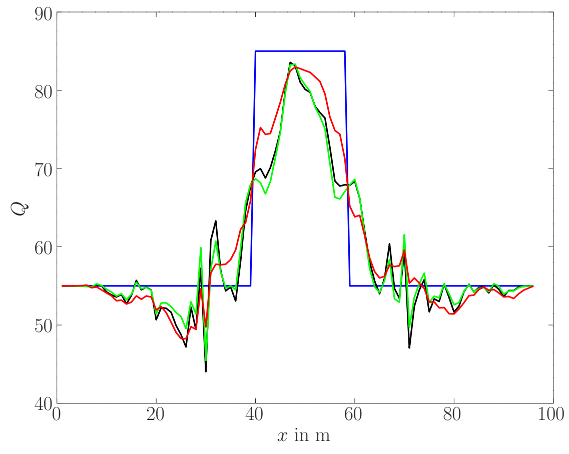
(b) p-wave velocity v_p



(c) s-wave velocity v_s



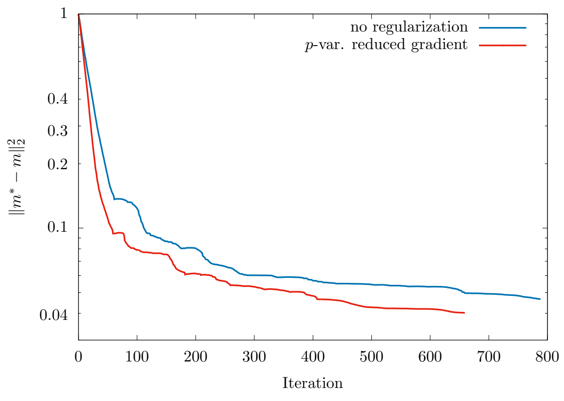
(d) Quality factor Q_1



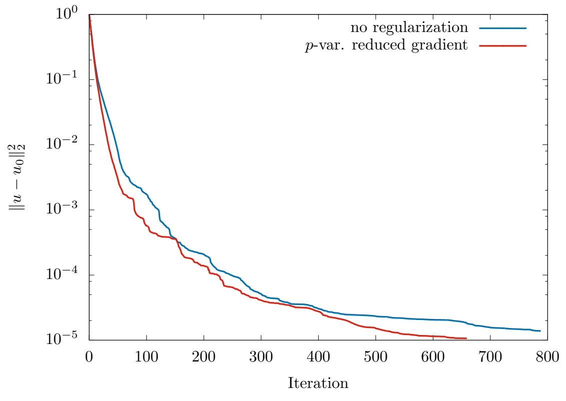
(e) Quality factor Q_2

Figure 5.24: Profiles of the simple example along $x = 50m$ using different regularization methods and perturbed objective data. The blue line represents the true model, the black line represents using no regularization, the green line represents using p-variation regularization and the red line represents using p-variation reduced gradient inversion.

inversion can reduce both the data misfit by 23 % compared to no regularization, and also reduce the model misfit by 13 %. Furthermore, the error reduction is faster for both misfits, as shown in figure 5.25.



(a) model misfit



(b) data misfit

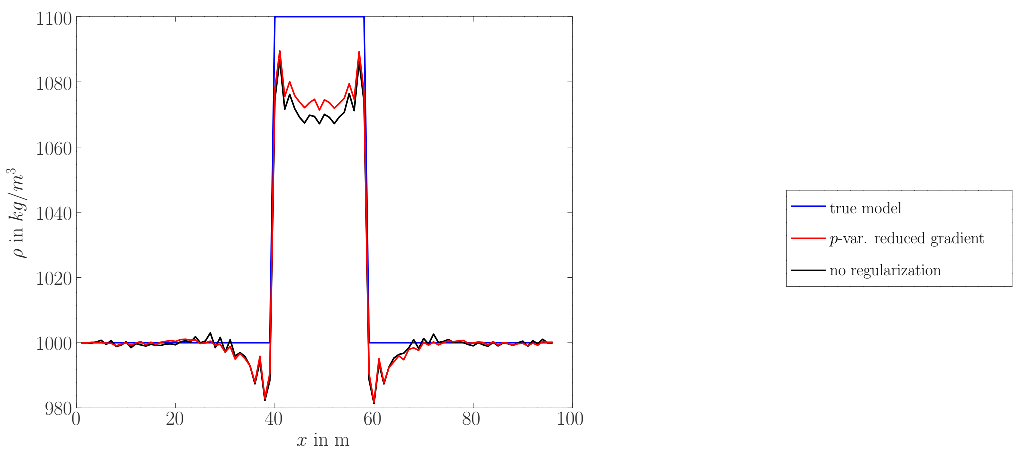
Figure 5.25: Convergence of the data and model misfit for the simple example using unperturbed objective data.

Considering the profiles along $x = 50m$ of the solutions, we see that p-variation reduced gradient inversion in particularly reduces the oscillations in the outer region, which can be seen in figures 5.26a and 5.26d. In addition, the density ρ is better approximated in the inner region (compare figure 5.26d).

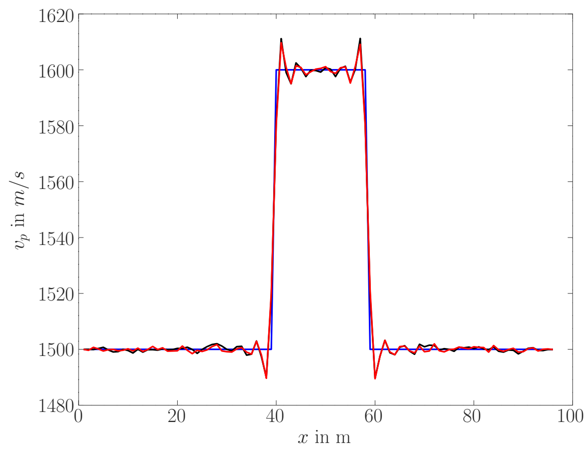
Findings and implications for the following sections The p-variation reduced gradient inversion leads to faster convergence of the data and model misfit compared to the other regularization methods presented. Likewise, the method achieves lower data and model misfit than the other regularization methods. Reducing the p-variation norm by integrating it directly into the inversion method, as in the p-variation reduced gradient inversion, yields better data and model misfits than realizing it only by using the p-variation regularization. In contrast to the other regularization methods, the p-variation reduced gradient inversion also achieves better results for unperturbed data than without regularization.

5.6.5 Multiscale inversion for viscoelastic modeling

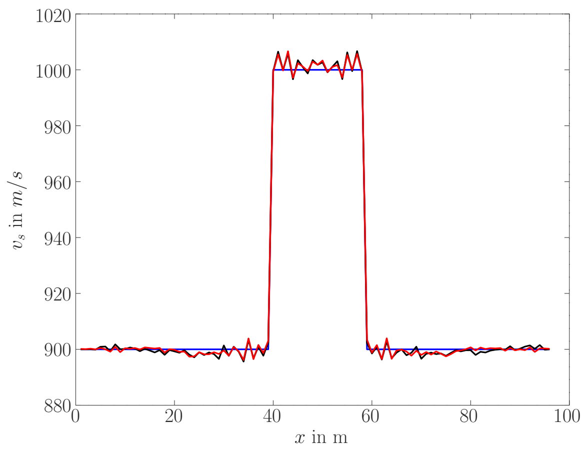
As described in subsection 5.5.3, multiscale approaches are usually necessary to invert complex models without getting stuck in local minima. In addition, information is lost in the transition to a low frequency model, so regularization can be helpful. Therefore, we want to investigate the influence of regularization and p-variation reduced gradient inversion on this approach in more detail. Based on the results in the previous sections, we compare no regularization, p-variation regularization, and the p-variation reduced gradient inversion with each other. For this we use viscoelastic PDE for the bubble example



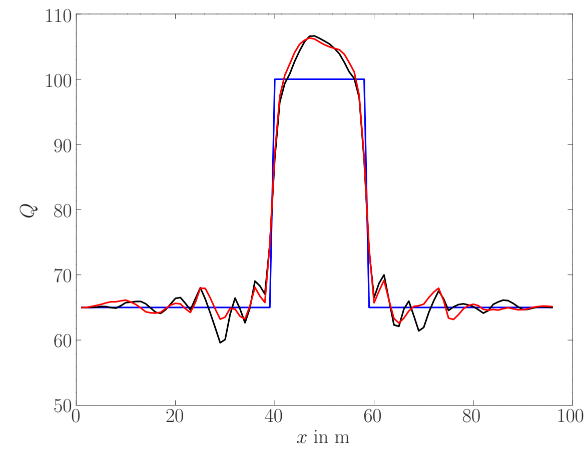
(a) Density ρ



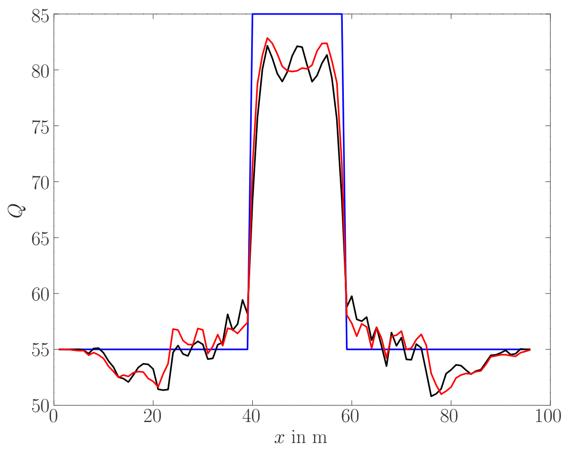
(b) p-wave velocity v_p



(c) s-wave velocity v_s



(d) Quality factor Q_1

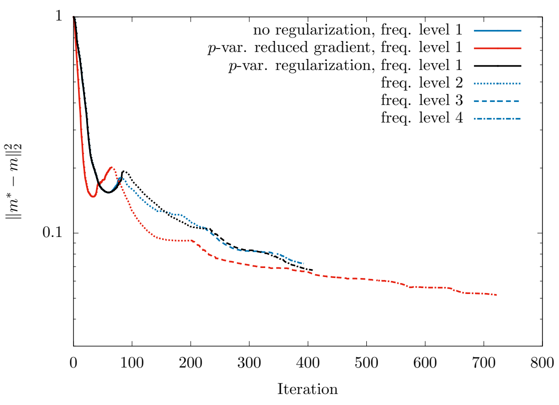


(e) Quality factor Q_2

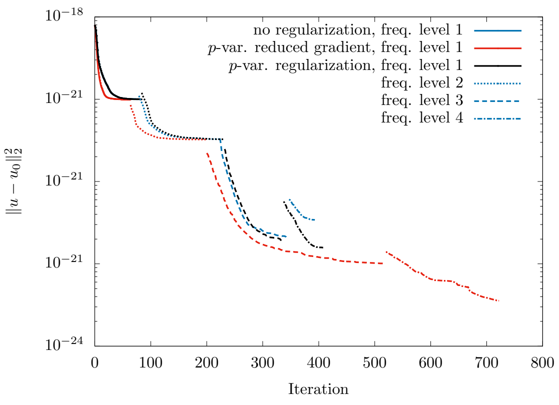
Figure 5.26: Profiles of the simple example along $x = 50m$ using different regularization methods and unperturbed objective data. The blue line represents the true model, the black line represents using no regularization and the red line represents using p-variation reduced gradient inversion.

with high attenuation and use four frequency levels $[0.01, 70]$, $[0.01, 100]$, $[0.01, 150]$ and $[0.01, 200]$.

For the low frequencies, the p-variation regularization yields the worst model misfit, but the lowest data misfit (cf. figures 5.27a and 5.27b). However, as described in section 5.5.3, the solution may have a worse model misfit even at the coarse level, since details cannot be inverted there. On the second frequency level, the p-variation reduced gradient inversion yields both better data and model misfit compared to the others, as shown by the misfit plots in figure 5.27. Overall, we obtain a 40 % smaller model misfit using p-variation reduced gradient inversion than using the standard inversion without any regularization and a 30 % smaller model misfit than using the standard inversion regularized by the p-variation regularization. Furthermore, figure 5.27 shows that the reduction of both the data misfit and the model misfit is faster for the p-variation reduced gradient inversion method.



(a) model misfit



(b) data misfit

Figure 5.27: Convergence of the data and model misfit for the simple example using the multiscale approach.

However, if we look at the profiles of the results for low frequency level (cf. figure 5.28 left column) we see that the Q factors cannot be approximated well. In particular, we obtain a Q factor that is too small in the inner region. We recall that the Q factor approximation is determined for an initially defined frequency band. By restricting to the low frequencies, not all of the frequency band on which Q is defined appears in the simulation. Thus, since not all frequencies are used in the inversion, the ratio between the unrelaxed modulus M_U and the relaxed M_R (cf. equation 5.71) changes for identical Q factor and different frequency levels. Therefore, a lower Q factor is assumed for lower frequencies, so that the same relation between M_U and M_R is obtained as for the total frequency band.

Overall, the p-variation reduced gradient inversion provides a better inversion of the Q factors and less oscillations in the outer region, analogous to the inversion in the whole frequency range.

Findings and implications for the following sections Using the multiscale approach, the inversion of the Q factors is significantly improved by the p-variation reduced gradient inversion. Furthermore, the p-variation reduced gradient inversion can smooth oscillations better than the considered regularization methods. Nevertheless, inversion of the Q factors is not possible accurately for low frequency bands.

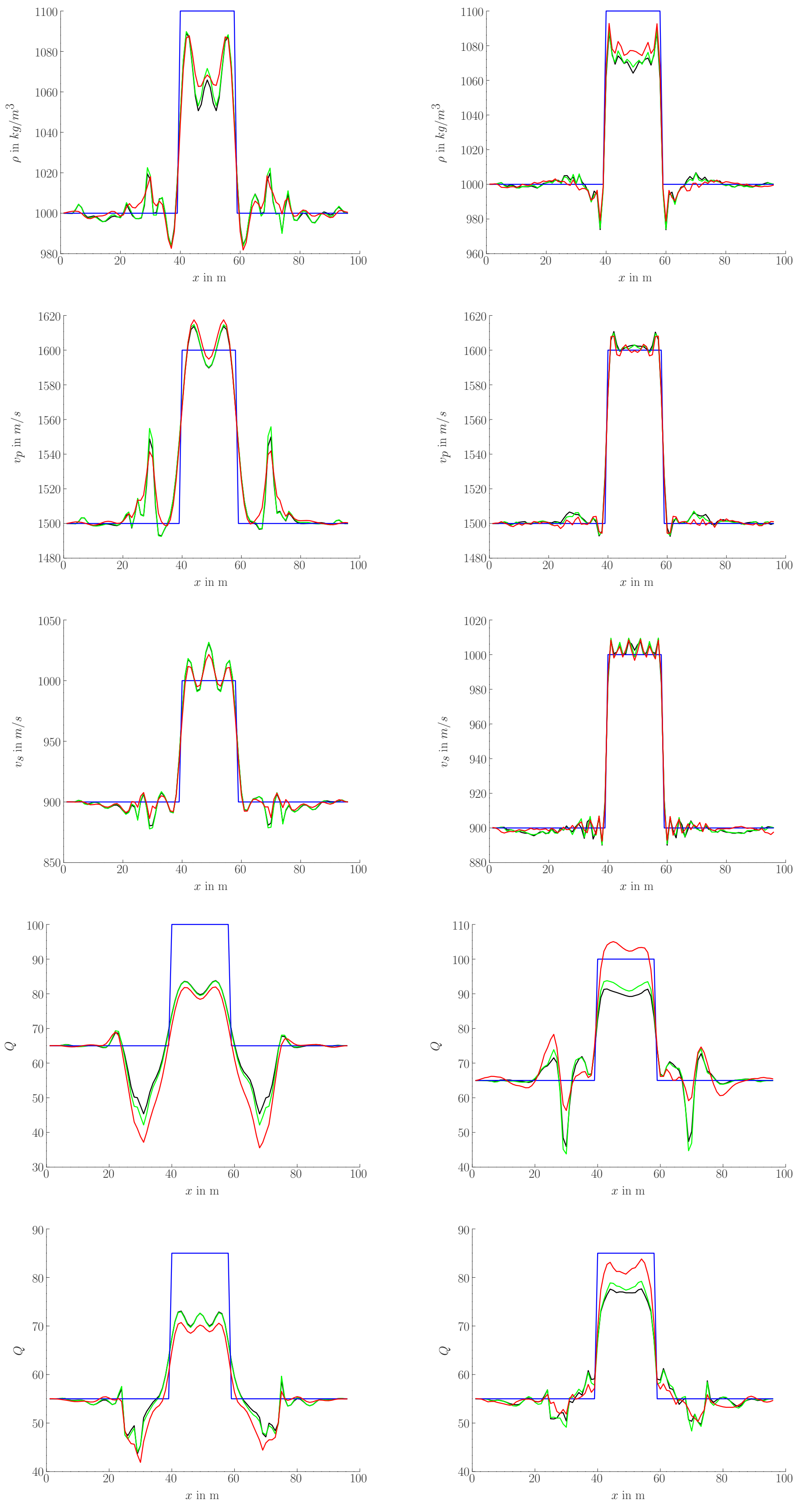
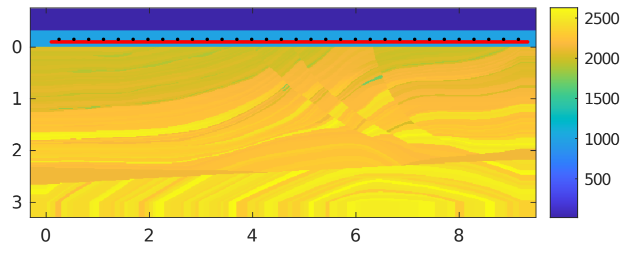


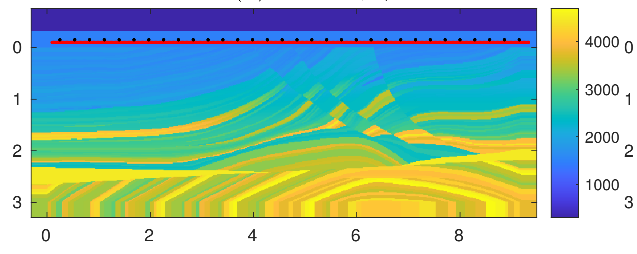
Figure 5.28: Profiles of the simple example along $x = 50m$ for multiscale inversion using no regularization (black), p-variation regularization (green) and p-variation reduced gradient inversion (red). The blue line indicates the true model. From top to bottom each row corresponds to ρ, v_p, v_s, Q_1 and Q_2. Left column are the results for the lowest frequency level, right column for the highest level.

5.6.6 Inversion of the Marmousi model

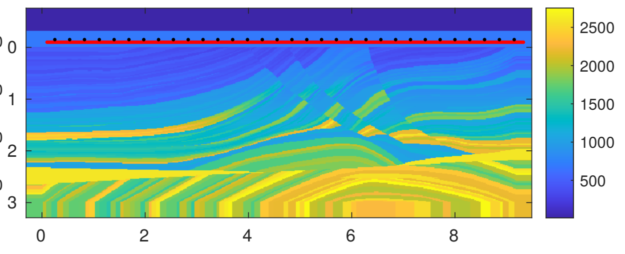
In this section we apply the p-variation reduced gradient inversion to the Marmousi example, which is established in geophysics. It is one of the standard benchmark problems and is used for example in [4, 80, 90, 102] and many other papers as an application benchmark. The original Marmousi model was built to resemble an overall continental drift geological setting. The geometry of the Marmousi model is based on a profile through the North Quenguela in the Cuanza basin [37, 161]. It was originally designed for the acoustic wave equation only. Therefore, we use here the so-called Marmousi2 model extended for the elastic wave equation, which was developed by Martin [102]. We extend this model and add the Q factors Q_1 and Q_2, based on [68]. A detailed description of how we generate the Q factors is given in the appendix in section A.1.8. This results in the model shown in figure 5.29, where the black points represent the 32 sources and the red points represent the 370 receivers. As with the bubble example before, the sources are shot sources that specify a pressure wave as the source time function, and the receivers are velocity meters. We use Ricker wavelets as source time functions with a dominant frequency of f_0=4, a time shift of $t_0 = 1.3$ and an amplitude of $A = 10^{10}$. A detailed description of the setup is given in the appendix in section A.1.8. For the modified Zener model, we use $N_{\text{SLS}} = 3$



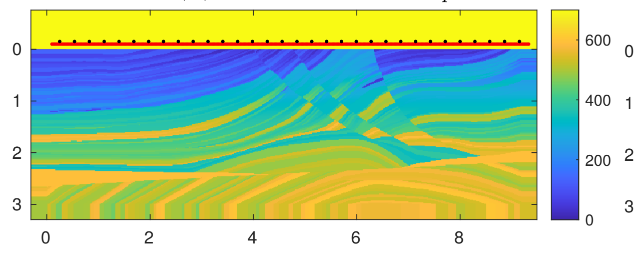
(a) Density ρ



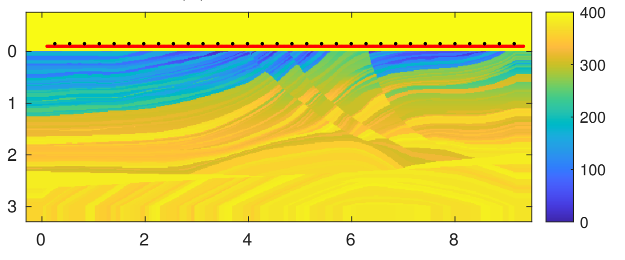
(b) p-wave velocity v_p



(c) s-wave velocity v_s



(d) Quality factor Q_1



(e) Quality factor Q_2

Figure 5.29: Marmousi2 reference/true model. The black points describe the positions of the sources, the red points describe the locations of the receivers. This setup is also used to create the objective data.

parallel Zener models, where we consider the objective function (5.67) in the frequency domain of $[\exp(\log(f_0) - \log(12) \cdot 0.5),\ 12 \cdot \exp(\log(f_0) - \log(12) \cdot 0.5)] \approx [28.87,\ 346.41]$ with $K = 12$ discrete frequencies. We set the Q ranges as follows $Q_1 \in [15,\ 650]$ and $Q_2 \in [15,\ 400]$.

Based on the results from the previous sections, we compare the standard inversion using no regularization with the p-variation reduced gradient inversion, where we also use the p-variation regularization. A basic regularization using the methods from the previous section provides only small or no improvement for this example compared to no regularization, so they are omitted here for the sake of clarity. We again consider perturbed and unperturbed objective data.

We use a multiscale approach with the following 5 frequency levels: $[0.01,\ 3]$, $[0.01,\ 5]$, $[0.01,\ 7]$, $[0.01,\ 9]$ and $[0.01,\ 10]$.

In addition, we set the number of iterations per level to 25, since we saw in the previous section that the error reduction is faster in the first 25–50 iterations (cf. figures 5.25 and 5.23). For a fair comparison, we set the number of iterations in both the p-variation reduced gradient inversion and the posterior standard inversion to 13.

Unperturbed objective data First, we consider the inversion of the viscoelastic PDE for the Marmousi model using unperturbed objective data. We compare the standard inversion without regularization with the p-variation reduced gradient inversion. Here we choose $\zeta = 1.05$ and $\kappa_{\mathcal{T}_p} = 5\text{E-}7$. The data misfit reduces slightly faster when using the p-variation reduced gradient inversion and the resulting data misfit is slightly lower (cf. figure 5.30a). The model misfit also reduces faster at each level, as shown in table 5.8, although the difference is very small at levels 2–4. Overall, the p-variation reduced gradient

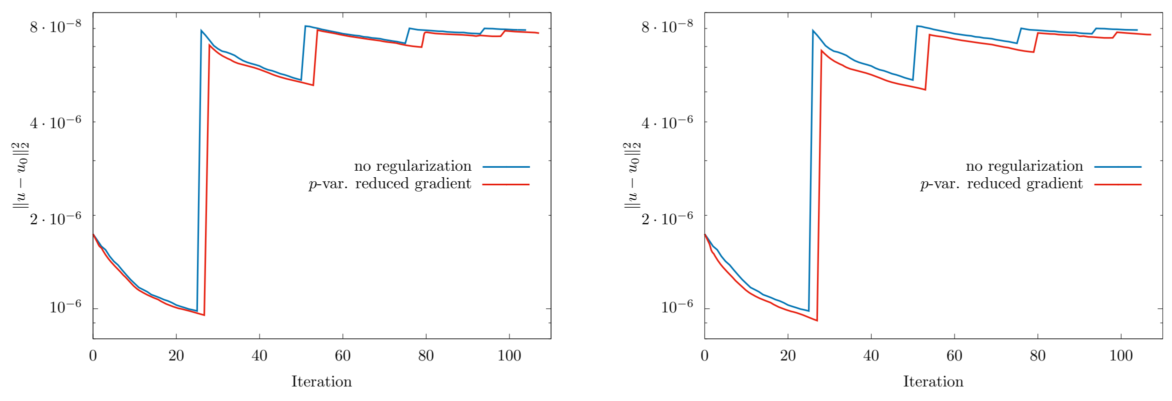
	1	2	3	4	5	total
no regularization	6.1E+9	1.4E+10	8.8E+9	1.4E+9	5.4E+9	3.57E+10
p-var. reduced gradient	6.2E+9	1.4E+10	8.8E+9	1.4E+9	5.5E+9	3.59E+10

Table 5.8: Model misfit reduction on each frequency level for the Marmousi model with perturbed data. The initial mode misfit is 1.06e+14. These large values result from the domain size and because the physical properties of the density and the velocities are in the order of 1E+3.

inversion yields a model misfit that is of order 2E+8 smaller. For instance, if we look at the profile line at $x = 4.5km$ (cf. figure 5.31), the modified inversion is better especially where regions with strongly different physical values interact. Furthermore, the modified method achieves a better inversion of the Q factors, although the inversion only works well for small Q factors. In general, both methods have difficulties in inverting deeper regions well, whereas the p-variation reduced gradient inversion achieves slightly better results.

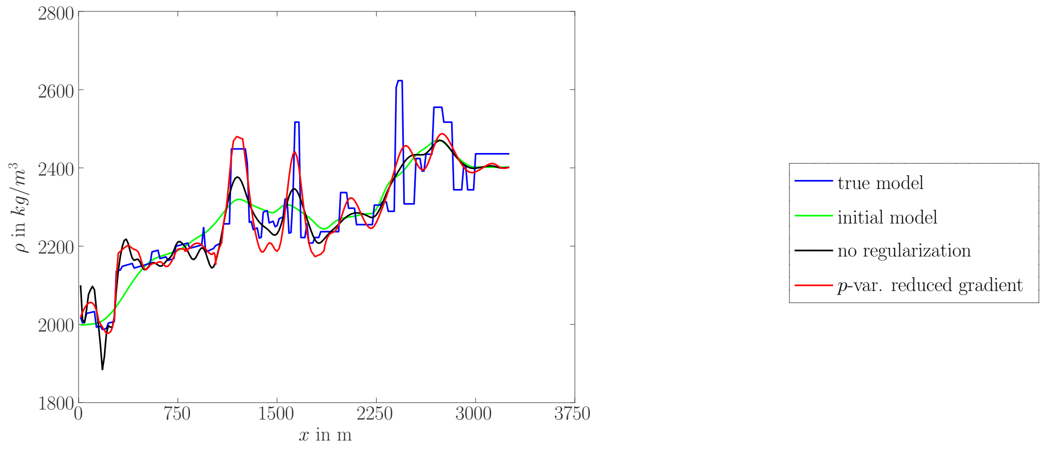
The s-wave velocity is inverted best. Here it can be seen again that the p-variation reduced gradient inversion gives better results in deeper regions and that regions adjacent to regions with strongly differing physical properties are inverted better.

The initial model as well as the resulting model using the modified inversion is shown in figure 5.32.

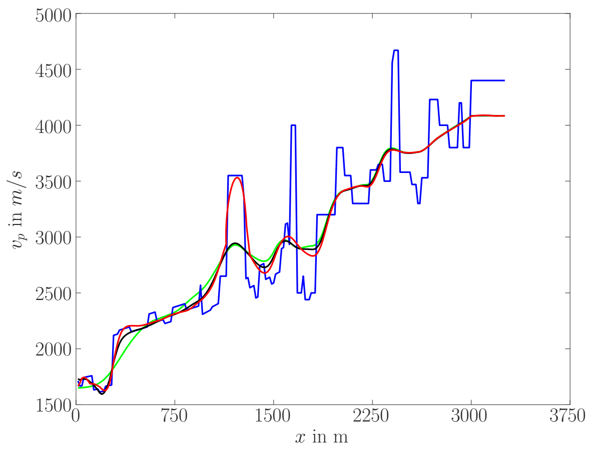


(a) model misfit (b) data misfit

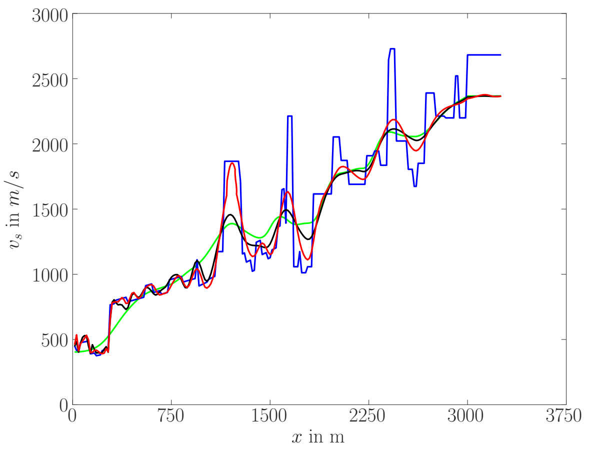
Figure 5.30: Reduction of the data misfit for the Marmousi example using unperturbed data (left) and perturbed data (right). The jumps are between the different frequency levels of the multiscale approach. The reduction is lower at higher frequency levels because the coarse structures are already well inverted and only details and finer regions are enhanced.



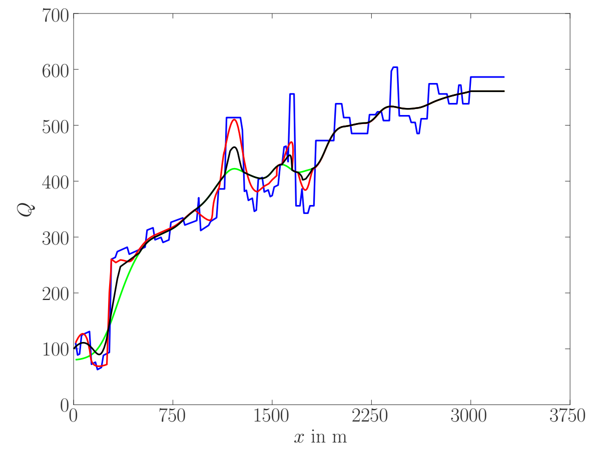
(a) Density ρ



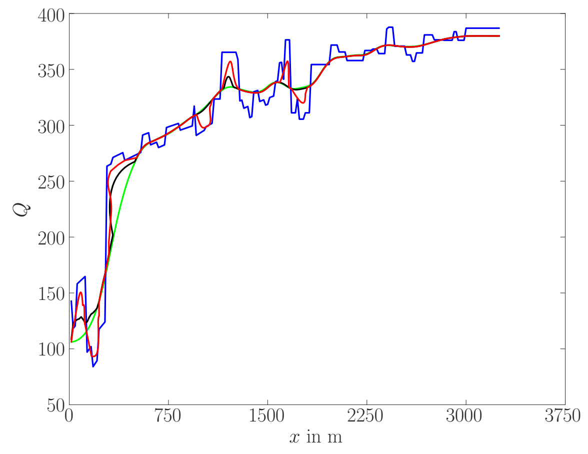
(b) p-wave velocity v_p



(c) s-wave velocity v_s



(d) Quality factor Q_1



(e) Quality factor Q_2

Figure 5.31: Profiles of the Marmousi model along $x = 4500m$, using unperturbed objective data.

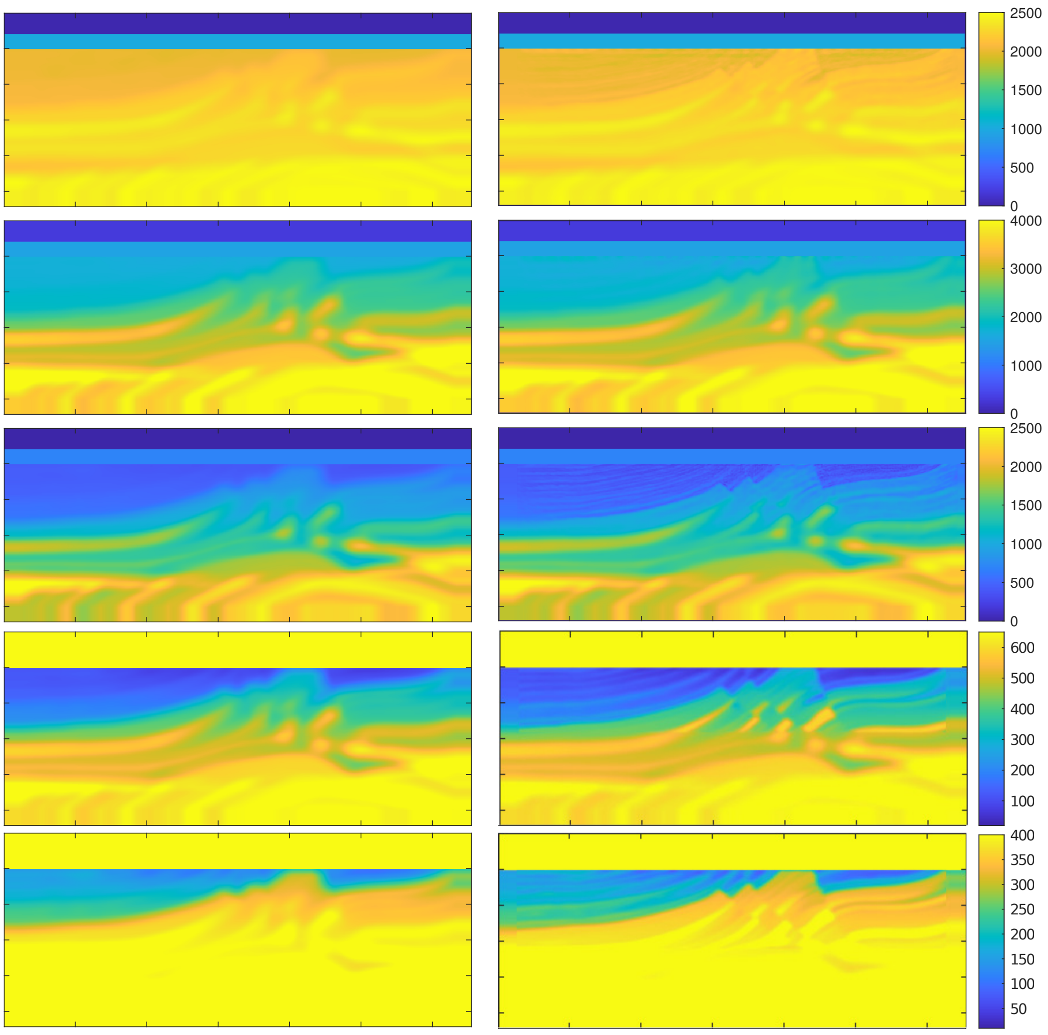


Figure 5.32: Inital (left) and resulting (right) model for the Marmousi example using the p-variation reduced gradient inversion.

Perturbed objective data For the simple bubble example, the improvement of the p-variation reduced gradient inversion had a larger influence on perturbed data than on unperturbed data. Whether this is also the case for the Marmousi example is examined in the following.

For this purpose, we add a normally distributed Gaussian noise with a variance of 10^{-12} to the initial data obtained by performing the forward simulation for the true model. This results in a signal to noise ratio SNR_{DB} between 34.5 and 52.5 dB. Similar to the inversion with unperturbed data, we choose $\zeta = 1.05$ and $\kappa_{\mathcal{T}_p} =$ 5E-7. Likewise, when using undisturbed data, using the p-variation reduced gradient inversion reduces the data misfit slightly faster and the resulting data misfit is slightly smaller (cf. figure 5.30b). The model misfit also reduces faster at each level, as shown in table 5.9, although the difference is very small at level 3. Comparing the data misfit reduction for the unperturbed

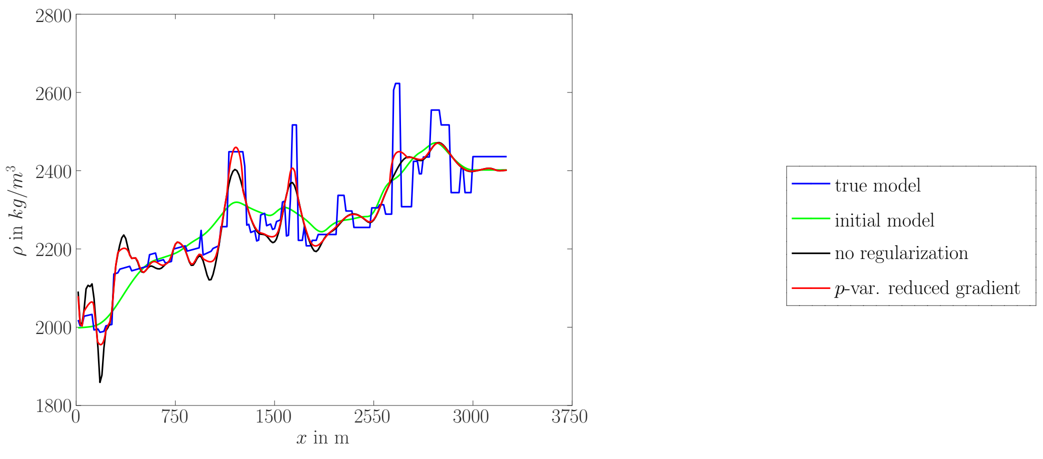
	1	2	3	4	5	total
no regularization	5.8E+9	1.3E+10	8.7E+9	1.4E+9	5.3E+9	34.2E+10
p-var. reduced gradient	6.0E+9	1.4E+10	8.7E+9	1.5E+9	5.4E+9	35.6E+10

Table 5.9: Model misfit reduction on each frequency level for the Marmousi model with perturbed data. The initial mode misfit is 1.06e+14. These large values result from the domain size and because the physical properties of the density and the velocities are in the order of 1E+3.

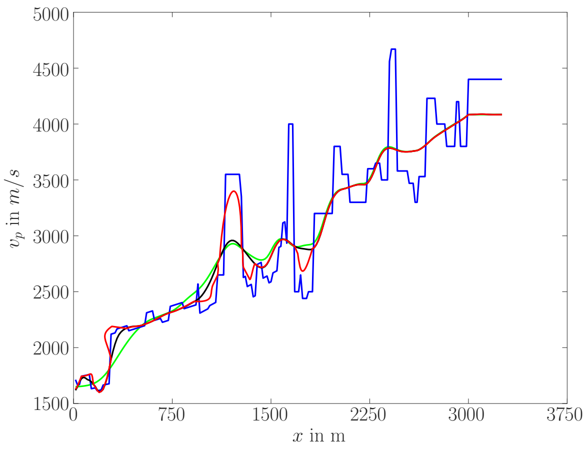
data (cf. figure 5.30a) with that for the perturbed data (cf. figure 5.30b), we see that the p-variation reduced gradient inversion gives more improvements for perturbed data, analogous to the simple bubble example. Likewise, the influence on model misfit is larger when using perturbed data (cf. tables 5.8 and 5.9).

Considering for instance the profile line at $x = 4.5km$ (cf. figure 5.33) shows the same differences as when using undisturbed data. The modified inversion method is especially better when regions with strongly different physical values meet each other. In addition, the modified method achieves a better inversion of the Q factors, although again inverting only works well for small Q factors. Overall, the inversion of the Q factors achieves slightly worse results than for unperturbed data. In general, both methods have even greater difficulty in inverting deeper regions well than for unperturbed data, where the p-variation reduced gradient inversion yields slightly better results.

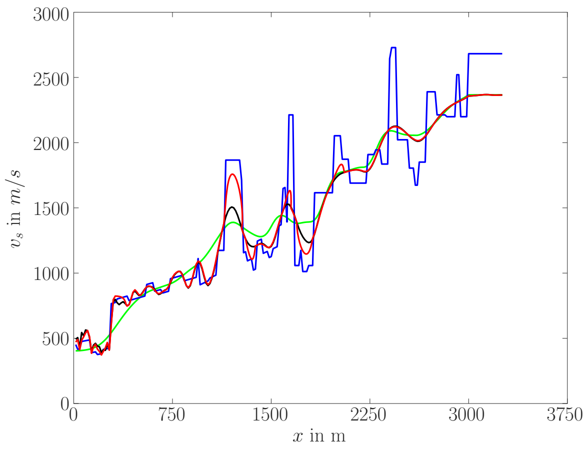
Findings The properties of the p-variation reduced gradient inversion resulting from the inversion of the bubble example were confirmed using the Marmousi example. The modified inversion method leads to improved model and data misfits compared to the standard method. In particular, for noisy data, the modified inversion achieves lower data and model misfits and accelerates the reduction of data and model misfits. The modified method achieves the strongest improvements in inverting deep regions, inverting Q factors, and inverting regions that are adjacent to regions with strongly different physical parameters. The improvements are especially noticeable with noisy data.



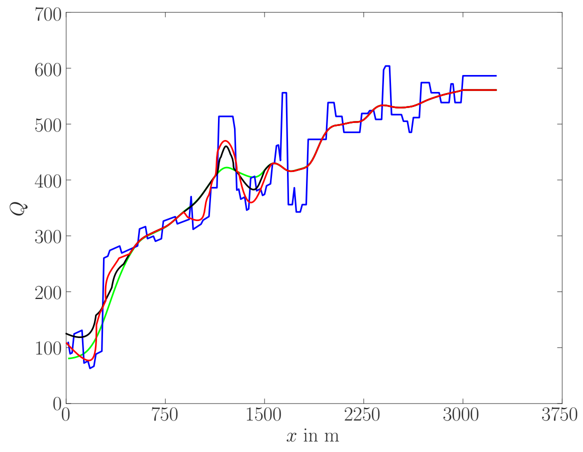
(a) Density ρ



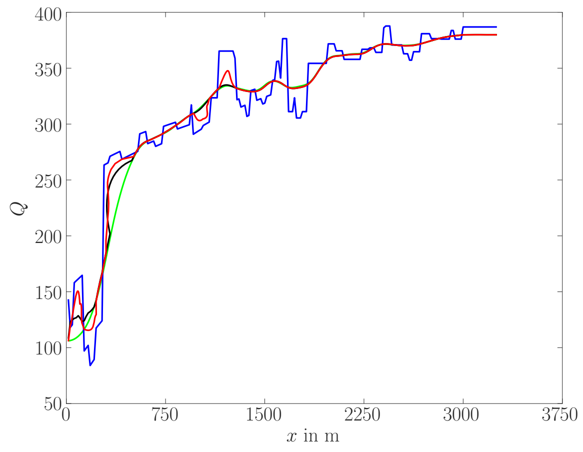
(b) p-wave velocity v_p



(c) s-wave velocity v_s



(d) Quality factor Q_1



(e) Quality factor Q_2

Figure 5.33: Profiles of the Marmousi model along $x = 4500m$, using perturbed objective data.

5.6.7 Strong and weak scaling

To evaluate the scalability of our FWI algorithm, we consider the bubble example with high attenuation (see figure 5.21 and table 5.1).

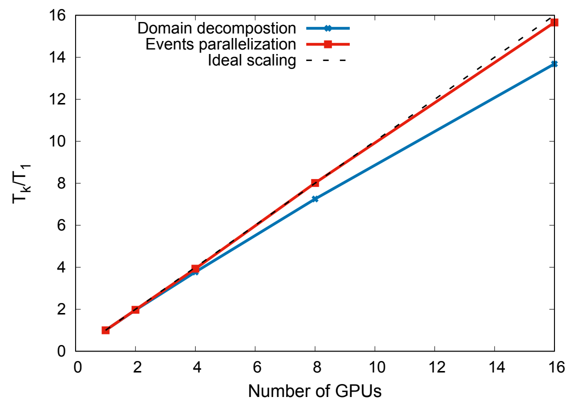
Hardware The scaling tests are performed on the BinAC cluster, a Tier-3 machine operated by the University of Tübingen as part of the BW-HPC strategy. This cluster has 62 GPU nodes, each contains two Intel Broadwell E5-2630v4 processors (base frequency of 2.2 GHz), 128 GB DDR4 RAM, and two NVIDIA Tesla K80. The K80 design comprises two identical Kepler GPUs per accelerator board, each with its dedicated GDDR memory, for a total of two times two GPUs per node. Ut presents the state of the arts as of 2016. We expect similar or even better scaling on new hardware. The use of finite differences leads to a very structured method, so that warp divergence is not an issue in the implementation (cf. section 2.2.2). Rather, an NVLINK between the GPUs would improve scalability, but in no case decrease it (see section 2.3). Therefore, we omit the use of more recent hardware in this section.

Weak scaling There are two different parallelization options as described in section 5.5.3. On the one hand the parallelization across the separate events and on the other hand by using domain decomposition. In order to be able to evaluate both types of parallelization, we consider them independently of each other.
Since the individual events are independent of each other, this parallelization is trivial. Communication only occurs when calculating the gradient of the objective function. Thus, depending on the number of GPUs, we also increase the number of events to be calculated, so for one GPU only one event is calculated, for two GPUs two events and so on. The weak scalability is almost ideal for this case (see table 5.10 and figure 5.34).

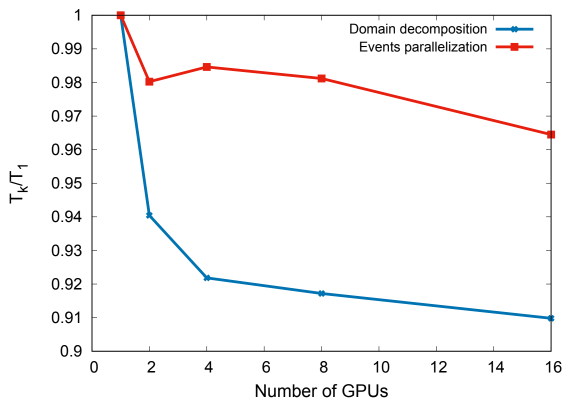
To address the weak scalability for the domain decomposition, we compute only one event and scale up the domain accordingly. We start with the standard $100m \times 100m$ area for one GPU, scale it up to an area of size $100m \times 200m$ for 2 GPUs, to an area of size $200m \times 200m$ for 4 GPUs, and so on. Weak scalability is slightly worse for this decomposition than for the parallelization of events, as the values from table 5.10 and figure 5.34b show.

#GPUs	domain decomposition	event decomposition
1	1.00	1.00
2	0.94	0.98
4	0.92	0.98
8	0.92	0.98
16	0.91	0.96

Table 5.10: Normalized runtimes of the weak scaling test.



(a) Strong scaling



(b) Weak scaling

Figure 5.34: Strong (left) and weak scaling (right) of the FWI algorithm 5.3.

Strong scaling As in the case of weak scalability, we also consider both types of parallelization independently of each other in the case of strong scalability. For the parallelization of the events we use the bubble example with the high attenuation with 16 events. Since the events represent independent forward and adjoint simulations and communication only occurs when calculating the gradient of the misfit function, we obtain an almost ideal strong scalability and the parallel efficiency is between 1 and 0.98 (cf. table 5.11 and figure 5.34).

We compute only one event to investigate the strong scalability for the domain decomposition and increase the area by a factor of 4 to have enough computational effort to perform the domain decomposition with 16 processes, i.e, we use an area of size $400m \times 400m$. As shown in figure 5.34a, the scalability in this case is slightly worse than when the events are parallelized, but still good with a parallel efficiency between 0.86 and 0.98.

	domain decomposition		event decomposition	
#GPUs	runtime	E	runtime	E
1	1.00		1.00	
2	1.96	0.98	1.97	0.99
4	3.77	0.94	3.93	0.98
8	7.25	0.91	8.01	1.00
16	13.68	0.86	15.66	0.98

Table 5.11: Normalized runtimes and parallel efficiency E of the strong scaling test.

Findings Event decomposition provides better strong and weak scalability and is preferable to area decomposition. However, the parallelization degree of the event decomposition is limited by the number of events. Domain decomposition also provides good strong and weak scalability, so it should be used in addition to event parallelization to increase the degree of parallelization. This means that the event decomposition should be used first, and if further parallelism is necessary or desired, the domain decomposition should be used in addition.

5.7 Conclusions

In this chapter, we have considered the seismic waveform modeling and inversion and derived an approach to invert the viscoelastic equation that allows inverting the Q factors as well. This derivation is based on an idea of Fichtner and van Driel [54] which directly inverts the Q factors and not the relaxation parameters. Furthermore, we added a piecewise linear penalty term in this approach to increase the accuracy of the Q approximation. Considering the viscoelastic equation increases the complexity of the simulation, so efficient inversion methods and implementations are necessary. Therefore, we have presented and compared different regularization methods. In addition, based on these results, we have presented a modified inversion method that reduces the p-variation in the gradient of the misfit function. Using a simple example, we have shown that the modified inverting method leads to faster error reductions in model and data misfits. Similarly, the resulting data and model misfits are lower when using this method. Afterwards, these findings were verified using the Marmousi example. In addition, it was shown that the inversion of the Q factors works better with the modified inversion method for the Marmousi model. Also, deeper regions, which are generally difficult to invert, can be inverted better with the p-variation reduced gradient inversion.

Through the implementation techniques presented in the previous chapters, we were also able to present a parallel GPU implementation that has shown good strong and weak scalability.

Regularization methods usually depend not only on the PDE to be inverted but also on the domain. Besides the Marmousi example, the so called saltdome example [10, 25] is another very common example on which the influence of the p-variation reduced gradient inversion can be studied. The presence of salt bodies is challenging for inverting because strong velocity contrasts between the salt and the surrounding sediments, the complex structure of the salt bodies, and rugose interfaces usually complicate seismic wave propagation and cause significant problems [132]. In addition, such regions usually consist of larger areas that are not as detailed as the Marmousi example. Furthermore, the combination of the p-variation reduced gradient inversion with different regularization methods can further improve the inversion. Similarly, the p-variation reduced gradient inversion approach can be adapted to other regularization methods, e.g., reduced $\|\Delta \cdot\|_2$ norm. This would allow any combination of stronger realization of the regularization properties in the inversion method and additionally in the regularization method. In this work, we have only considered the combination of p-variation reduction in the inversion method and in the regularization.

Furthermore, better inversion of the Q factors at lower frequency levels using the multiscale approach would be desirable to further improve convergence.

6

Summary

In this thesis, the implementation of efficient simulations for challenging PDE problems was discussed. On the basis of three different applications, implementations for different platforms were realized, ranging from high performance workstations to small clusters and up to super computers.

In the first part of this thesis different implementation techniques were developed to increase the efficiency of numerical PDE simulation software. In particular, the focus was on improvements of the GPU implementation. In chapter 3 and 4 many of the challenges described in chapter 2 were addressed and overcome. The implementation techniques presented can be applied to many other (unstructured) PDE simulations. Among others, in chapter 4 a modified implementation of the SPH method based on a pairs neighbor list was presented. Included is a modified NNS algorithm, which returns the neighborhood relations in a pairs list. This modification leads to a significant increase in efficiency on current hardware. Furthermore, different communication strategies were described and compared in chapter 3.

In the second part of this thesis the accuracy of the waveform modeling and inversion was improved. The elastic wave equation was transformed into a viscoelastic wave equation, where the inversion of the Q factors is feasible. However, this approach results in a system of PDEs that obtains significantly more equations than in the elastic case, and thus the simulation requires more memory and computational effort. To solve such simulations in acceptable time requires powerful numerical methods and efficient implementations. The efficient implementation techniques were already addressed in the chapters 3 and 4, so in chapter 5 the focus was on the improvements of the accuracy of the methods.

An improvement of the Q factor approximation was described, which was achieved by a piecewise linear penalty term. Another contribution is the development of a modified inverting method based on a p-variation reduced gradient. This method leads to lower misfits and faster convergence for two studied examples, compared to other known regularization methods.

There are, of course, still several open questions and various topics for further research.

The future availability of MPI-GDS [159], could improve the asynchronous communication in our RK4IP implementation in chapter 3. In addition, also the optimization of collective operations which are under investigation [17, 18] could improve MPI communications in the RK4IP and SPH implementation.

Furthermore, in chapter 5 we considered only two exemplary examples. However, regularization methods usually depend not only on the PDE to be inverted but also on the domain. Besides the Marmousi example, the so called saltdome example [10, 25] is another very common example on which the influence of the p-variation reduced gradient inversion can be studied. Furthermore, the combination of the p-variation reduced gradient inversion with different regularization methods can further improve the inversion. Additionally, a better inversion of the Q factors at lower frequency levels using the multiscale approach would be helpful to further improve the convergence.

Bibliography

[1] S. Adami, X. Y. Hu, and N. A. Adams. A generalized wall boundary condition for smoothed particle hydrodynamics. *Journal of Computational Physics*, 231(21):7057–7075, 2012.

[2] S. Adami, X. Y. Hu, and N. A. Adams. A transport-velocity formulation for smoothed particle hydrodynamics. *Journal of Computational Physics*, 241:292 – 307, 2013. DOI: `10.1016/j.jcp.2013.01.043`.

[3] A. Adinets. Optimized filtering with warp-aggregated atomics, 2014. URL: `https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/`.

[4] H. S. Aghamiry, A. Gholami, and S. Operto. Full waveform inversion by proximal Newton method using adaptive regularization. *Geophysical Journal International*, 224(1):169–180, 2020. DOI: `10.1093/gji/ggaa434`.

[5] G. P. Agrawal. *Nonlinear Fiber Optics*. Academic Press, 5th edition, 2012.

[6] J. M. Alcaraz-Pelegrina and P. Rodríguez-García. Simulations of pulse propagation in optical fibers using graphics processor units. *Computer Physics Communications*, 182(7):1414–1420, 2011. DOI: `10.1016/j.cpc.2011.03.007`.

[7] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford Science Publications. Clarendon Press, 1987.

[8] G. Almasi. *PGAS (Partitioned Global Address Space) Languages*. In *Encyclopedia of Parallel Computing*. D. Padua, editor. Springer US, Boston, MA, 2011, pages 1539–1545. DOI: `10.1007/978-0-387-09766-4_210`.

[9] A. Amara, F. Amiel, and T. Ea. FPGA vs. ASIC for low power applications. *Microelectronics Journal*, 37(8):669 –677, 2006. DOI: `10.1016/j.mejo.2005.11.003`.

[10] F. Aminzadeh, B. Jean, and T. Kunz. *3-D Salt and Overthrust Models*. Society of Exploration Geophysicists, 1997. ISBN: 978-1-560-80077-4.

[11] H. Ammari, E. Bretin, J. Garnier, and A. Wahab. Time-reversal algorithms in viscoelastic media. *European Journal of Applied Mathematics*, 24(4):565–600, 2013. DOI: `10.1017/S0956792513000107`.

[12] E. Hairer an G. Wanner and S. P. Nørsett. *Solving Ordinary Differential Equations I*. Springer Berlin Heidelberg, 1993. DOI: `10.1007/978-3-540-78862-1`.

[13] J. A. Anderson, J. Glaser, and S. C. Glotzer. HOOMD-blue: a python package for high-performance molecular dynamics and hard particle Monte Carlo simulations. *Computational Materials Science*, 173:109363, 2020. DOI: `10.1016/j.commatsci.2019.109363`.

[14] J. A. Anderson, D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[15] C. Antonelli, M. Shtaif, and A. Mecozzi. Modeling of Nonlinear Propagation in Space-Division Multiplexed Fiber-Optic Transmission. *Journal of Lightwave Technology*, 34(1):36–54, 2016.

[16] R. Aris. *Vectors, Tensors and the Basic Equations of Fluid Mechanics*. Dover Books on Mathematics. Dover Publications, 1990.

[17] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda. Optimized Broadcast for Deep Learning Workloads on Dense-GPU InfiniBand Clusters. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–9, Barcelona, Spain, 2018. DOI: `10.1145/3236367.3236381`.

[18] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda. Efficient Large Message Broadcast using NCCL and CUDA-Aware MPI for Deep Learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pages 15–22, Edinburgh, Scotland, 2016. DOI: `10.1145/2966884.2966912`.

[19] S. Balac and F. Mahé. Embedded Runge-Kutta scheme for step-size control in the interaction picture method. *Computer Physics Communications*, 184(4):1211–1219, 2013.

[20] A. Bamberger, Guy Chavent, and P. Lailly. About the stability of the inverse problem in 1-d wave equations application to the interpretation of seismic profiles. *Applied Mathematics and Optimization*, 5:1–47, 1979. DOI: `10.1007/BF01442542`.

[21] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library. Cambridge University Press, 2000. DOI: `10.1017/CBO9780511800955`.

[22] S. Beller, V. Monteiller, L. Combe, S. Operto, and G. Nolet. On the sensitivity of teleseismic full-waveform inversion to earth parametrization, initial model and acquisition design. *Geophysical Journal International*, 212(2):1344–1368, 2017. DOI: `10.1093/gji/ggx480`.

[23] J.-P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185 –200, 1994. DOI: `10.1006/jcph.1994.1159`.

[24] S. Bernard, V. Monteiller, D. Komatitsch, and P. Lasaygues. Ultrasonic computed tomography based on full-waveform inversion for bone quantitative imaging. *Physics in medicine and biology*, 62 17:7011–7035, 2017.

[25] F. J. Billette and S. Brandsberg-Dahl. The 2004 BP velocity benchmark. In *67th EAGE Conference & Exhibition*. European Association of Geoscientists & Engineers, 2005. DOI: `10.3997/2214-4609-pdb.1.b035`.

[26] É. Blanc, D. Komatitsch, E. Chaljub, B. Lombard, and Z. Xie. Highly accurate stability-preserving optimization of the Zener viscoelastic model, with application to wave propagation in the presence of strong attenuation. *Geophysical Journal International*, 205(1):427–439, 2016. DOI: `10.1093/gji/ggw024`.

[27] J. O. Blanch, J. O. A. Robertsson, and W. W. Symes. Viscoelastic finite-difference modeling. Technical report, Department of Computational and Applied Mathematics, Rice University, 1993.

[28] T. Bodurov. Derivation of the nonlinear Schrödinger equation from first principles. In *Annales de la Fondation Louis de Broglie*, volume 30 of number 3-4, pages 343–352. Fondation Louis de Broglie, 2005.

[29] D. Borisov, R. Modrak, F. Gao, and J. Tromp. 3d elastic full-waveform inversion of surface waves in the presence of irregular topography using an envelope-based misfit function. *GEOPHYSICS*, 83(1):R1–R11, 2018. DOI: `10.1190/geo2017-0081.1`.

[30] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3:1–122, 2011. DOI: `10.1561/2200000016`.

[31] E. Bozdağ, J. Trampert, and J. Tromp. Misfit functions for full waveform inversion based on instantaneous phase and envelope measurements. *Geophysical Journal International*, 185(2):845–870, 2011. DOI: `10.1111/j.1365-246x.2011.04970.x`.

[32] M. Brehler and P. M. Krummrich. Impact of WDM Channel Count on Nonlinear Effects in MDM Transmission Systems. In *Optical Fiber Communication Conference (OFC)*, Los Angeles, CA, USA, 2017. DOI: `10.1364/OFC.2017.Th2A.63`. paper Th2A.63.

[33] M. Brehler, M. Schirwon, D. Göddeke, and P. M. Krummrich. A GPU-Accelerated Fourth-Order Runge-Kutta in the Interaction Picture Method for the Simulation of Nonlinear Signal Propagation in Multimode Fibers. *Journal of Lightwave Technology*, 35(17):3622–3628, 2017. DOI: `10.1109/JLT.2017.2715358`.

[34] M. Brehler, M. Schirwon, D. Göddeke, and P. M. Krummrich. Modeling the Kerr-Nonlinearity in Mode-Division Multiplexing Fiber Transmission Systems on GPUs. In *Advanced Photonics Congress, Nonlinear Photonics (NP)*, Zurich, Switzerland, 2018. DOI: `10.1364/BGPPM.2018.JTu5A.27`. paper JTu5A.27.

[35] M. Brehler, M. Schirwon, P. M. Krummrich, and D. Göddeke. Simulation of nonlinear signal propagation in multimode fibers on multi-GPU systems. *Communications in Nonlinear Science and Numerical Simulation*, 84:105150, 2020.

[36] L. Brookshaw. A method of calculating radiative heat diffusion in particle simulations. *Publications of the Astronomical Society of Australia*, 6(2):207–210, 1985. DOI: 10.1017/S1323358000018117.

[37] A. Brougois, M. Bourget, P. Lailly, M. Poulet, P. Ricarte, and R. Versteeg. Marmousi, model and data. In 1990. DOI: 10.3997/2214-4609.201411190.

[38] C. G. BROYDEN. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970. DOI: 10.1093/imamat/6.1.76.

[39] C. Bunks, F. M. Saleck, S. Zaleski, and G. Chavent. Multiscale seismic waveform inversion. *GEOPHYSICS*, 60(5):1457–1473, 1995. DOI: 10.1190/1.1443880.

[40] J. M. Carcione. *Wave Fields in Real Media: Wave Propagation in Anisotropic, Anelastic, Porous and Electromagnetic Media*. Elsevier Science, 2007.

[41] A. Cevahir, A. Nukada, and S. Matsuoka. Fast conjugate gradients with multiple GPUs. In pages 893–903, 2009. DOI: 10.1007/978-3-642-01970-8_90.

[42] G. Chavent. Identification of function parameters in partial differential equations. In R. E. Goodson and M. Polis, editors, *Identification of parameter distributed systems*, pages 31–48. American Society of Mechanical Engineers, 1974.

[43] M. Christen, O. Schenk, and Y. Cui. Patus for convenient high-performance stencils: evaluation in earthquake simulations. In pages 1–10, 2012. DOI: 10.1109/SC.2012.95.

[44] B. Cloutier, B.K. Muite, and P. Rigge. Performance of FORTRAN and C GPU Extensions for a Benchmark Suite of Fourier Pseudospectral Algorithms. In *Symp. on Application Accelerators in High Performance Computing (SAAHPC)*, pages 145–148, Lemont, IL, USA, 2012.

[45] NVIDIA Corporation. NVIDIA T4 70W low profile PCIe GPU accelerator, 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-product-brief.pdf.

[46] J. Crank and P. Nicolson. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Mathematical Proceedings of the Cambridge Philosophical Society*, 43(1):50–67, 1947. DOI: 10.1017/S0305004100023197.

[47] E. Crase, A. Pica, M. Noble, J. McDonald, and A. Tarantola. Robust elastic nonlinear waveform inversion: application to real data. *GEOPHYSICS*, 55(5):527–538, 1990. DOI: 10.1190/1.1442864.

[48] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra. Parsec in practice: optimizing a legacy chemistry application through distributed task-based execution. In *2015 IEEE International Conference on Cluster Computing*, pages 304–313, 2015. DOI: 10.1109/CLUSTER.2015.50.

[49] R. Dolbeau. Theoretical peak flops per instruction set: a tutorial. *The Journal of Supercomputing*, 74, November 2017. DOI: 10.1007/s11227-017-2177-5.

[50] V. Dolean, P. Jolivet, and F. Nataf. *An Introduction to Domain Decomposition Methods*. Society for Industrial and Applied Mathematics, 2015. DOI: 10.1137/1.9781611974065.

[51] J. Dongarra, J. Hittinger, J. Bell, L. Chacón, R. Falgout, M. Heroux, P. Howland, E. Ng, C. Webster, S. Wild, and K. Pau. Applied Mathematics Research for Exascale Computing. Technical report, U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, 2014.

[52] G. Fabien-Ouellet, E. Gloaguen, and B. Giroux. Time domain viscoelastic full waveform inversion. *Geophysical Journal International*, 209:1718–1734, 2017. DOI: 10.1093/gji/ggx110.

[53] A. Fichtner. *Full Seismic Waveform Modelling and Inversion*. 2011. DOI: 10.1007/978-3-642-15807-0.

[54] A. Fichtner and M. van Driel. Models and Fréchet kernels for frequency-(in)dependent Q. *Geophysical Journal International*, 198(3):1878–1889, 2014. DOI: 10.1093/gji/ggu228.

[55] R. Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322, 1970. DOI: 10.1093/comjnl/13.3.317.

[56] D. Frenkel and B. Smit. *Understanding Molecular Simulation*. Elsevier, 2002. DOI: 10.1016/b978-0-12-267351-1.x5000-7.

[57] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[58] K. Gao and L. Huang. Acoustic- and elastic-waveform inversion with total generalized p-variation regularization. *Geophysical Journal International*, 218(2):933–957, 2019. DOI: 10.1093/gji/ggz203.

[59] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977. DOI: 10.1093/mnras/181.3.375.

[60] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications*, 192:97 –107, 2015. DOI: 10.1016/j.cpc.2015.02.028.

[61] J. Glaser, T. D. Nguyen, J. A. Anderson, F. Spiga P. Liu, J. A. Millan, D. C. Morse, and S. C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications*, 192:97–107, 2015.

[62] D. Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24(109):23–23, 1970. DOI: `10.1090/s0025-5718-1970-0258249-6`.

[63] T. Goldstein and S. Osher. The split Bregman method for $l1$-regularized problems. *SIAM Journal on Imaging Sciences*, 2(2):323–343, 2009. DOI: `10.1137/080725891`.

[64] Green500 — TOP500 Supercomputer Sites. URL: `https://www.top500.org/lists/green500/2020/11/`.

[65] L. Guasch, O. C. Agudo, M.-X. Tang, P. Nachev, and M. Warner. Full-waveform inversion imaging of the human brain. *npj Digital Medicine*, 3(1), 2020. DOI: `10.1038/s41746-020-0240-8`.

[66] C. Gélis, J. Virieux, and G. Grandjean. Two-dimensional elastic full waveform inversion using Born and Rytov formulations in the frequency domain. *Geophysical Journal International*, 168(2):605–633, 2007. DOI: `10.1111/j.1365-246X.2006.03135.x`.

[67] D. Göddeke and M. Schirwon. GPU programming with CUDA. SPPEXA Doctoral Retreat 2016, 2016.

[68] E. L. Hamilton. COMPRESSIONAL-WAVE ATTENUATION IN MARINE SEDIMENTS. *GEOPHYSICS*, 37(4):620–646, 1972. DOI: `10.1190/1.1440287`.

[69] A. M. Heidt. Efficient Adaptive Step Size Method for the Simulation of Supercontinuum Generation in Optical Fibers. *Journal of Lightwave Technology*, 27(18):3984–3991, 2009.

[70] S. Hellerbrand and N. Hanik. Fast Implementation of the Split-Step Fourier Method Using a Graphics Processing Unit. In *Optical Fiber Communication Conference (OFC)*, San Diego, CA, USA, 2010. DOI: `10.1364/OFC.2010.OTuD7`.

[71] M. A. Heroux. Software challenges for extreme scale computing: going from petascale to exascale systems. *International Journal of High Performance Computing Applications*, 23(4):437–439, 2009. DOI: `10.1177/1094342009347711`.

[72] N. J. Higham. *Accuracy and Stability of Numerical Algorithms - Second Edition.* SIAM, Philadelphia, 2002.

[73] R. Hoetzlein. Fast fixed-radius nearest neighbors: interactive million–particle fluids. *GPU Technology Conference*, 2014. URL: `https://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf`.

[74] M. P. Howard, J. A. Anderson, A. Nikoubashman, S. C. Glotzer, and A. Z. Panagiotopoulos. Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units. *Computer Physics Communications*, 203(Supplement C):45 –52, 2016. DOI: `10.1016/j.cpc.2016.02.003`.

[75] M. P. Howard, A. Statt, F. Madutsa, T. M. Truskett, and A. Z. Panagiotopoulos. Quantized bounding volume hierarchies for neighbor search in molecular simulations on graphics processing units. *Computational Materials Science*, 164:139 –146, 2019. DOI: `10.1016/j.commatsci.2019.04.004`.

[76] J. Hult. A Fourth-Order Runge-Kutta in the Interaction Picture Method for Simulating Supercontinuum Generation in Optical Fibers. *Journal of Lightwave Technology*, 25(12):3770–3775, 2007.

[77] A. Hutcheson and V. Natoli. Memory Bound vs . Compute Bound : A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications, 2011.

[78] P. J. in 't Veld, S. J. Plimpton, and G. S. Grest. Accurate and efficient methods for modeling colloidal mixtures in an explicit solvent using molecular dynamics. *Computer Physics Communications*, 179(5):320 –329, 2008. DOI: `10.1016/j.cpc.2008.03.005`.

[79] V. Isakov. *Inverse Problems for Partial Differential Equations*, volume 127. 2017. DOI: `10.1007/978-3-319-51658-5`.

[80] I. Ivanov, M. Belishev, and V. Semenov. The reconstruction of sound speed in the marmousi model by the boundary control method, 2016.

[81] Khronos OpenCL Working Group. The OpenCL C 3.0 specification, 2019. URL: `https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf`.

[82] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[83] D. Komatitsch and R. Martin. An unsplit convolutional Perfectly Matched Layer improved at grazing incidence for the seismic wave equation. *Society of Exploration Geophysicists*, 72(5):SM155–SM167, 2007. DOI: `10.1190/1.2757586`.

[84] D. Komatitsch, Z. Xie, E. Bozdağ, E Sales de Andrade, D. Peter, Q. Liu, and J. Tromp. Anelastic sensitivity kernels with parsimonious storage for adjoint tomography and full waveform inversion. *Geophysical Journal International*, 206(3):1467–1478, 2016. DOI: `10.1093/gji/ggw224`.

[85] J. Kraus. An Introduction to CUDA-Aware MPI, 2013. URL: `https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/`.

[86] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing*, 36, 2013. DOI: `10.1137/130930352`.

[87] L. Krischer, A. Fichtner, C. Boehm, and H. Igel. Automated large-scale full seismic waveform inversion for north america and the north atlantic. *Journal of Geophysical Research: Solid Earth*, 123(7):5902–5928, 2018. DOI: `10.1029/2017jb015289`.

[88] J. Kristek, P. Moczo, and M. Galis. A brief summary of some PML formulations and discretizations for the velocity-stress equation of seismic motion. *Studia Geophysica et Geodaetica*, 53(4):459–474, 2009. DOI: `10.1007/s11200-009-0034-6`.

[89] A. Kuntsevich and F. Kappel. Solvopt: the solver for local nonlinear optimization problems. *University of Graz*, 1997. DOI: `10.13140/RG.2.2.10451.43044`.

[90] A. Kurzmann, A. Przebindowska, D. Köhn, and T. Bohlen. Acoustic full waveform tomography in the presence of attenuation: a sensitivity analysis. *Geophysical Journal International*, 195(2):985–1000, 2013. DOI: `10.1093/gji/ggt305`.

[91] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 29–38, 2014. DOI: `10.1109/IPDPSW.2014.9`.

[92] P. Lailly. The seismic inverse problem as a sequence of before-stack migrations. In J. B. Bednar, R. Redner, E. Robinson, and A. Weglein, editors, *Proceedings of the Conference on Inverse Scattering, Theory and Application Expanded Abstracts*, pages 206–220. Society of Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[93] Y. Lin and V. Grover. Using cuda warp-level primitives, 2018. URL: `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`.

[94] Y. Lin and L. Huang. Acoustic- and elastic-waveform inversion using a modified total-variation regularization scheme. *Geophysical Journal International*, 200:489–502, 2014. DOI: `10.1093/gji/ggu393`.

[95] M. B. Liu and G. R. Liu. Smoothed particle hydrodynamics (SPH): an overview and recent developments. *Archives of computational methods in engineering*, 17(1):25–76, 2010.

[96] Q. Liu and J. Tromp. Finite-frequency kernels based on adjoint methods. *The Seismological Society of America*, 96(6):2383–2397, 2006. DOI: `10.1785/0120060041`.

[97] Q. Liu and J. Tromp. Finite-frequency sensitivity kernels for global seismic wave propagation based upon adjoint methods. *Geophysical Journal International*, 174(1):265–286, 2008. DOI: `10.1111/j.1365-246X.2008.03798.x`.

[98] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024, 1977. DOI: `10.1086/112164`.

[99] Q. Lui and J. Tromp. Finite-frequency kernels based on adjoint methods. *Bulletin of The Seismological Society of America - BULL SEISMOL SOC AMER*, 96:2383–2397, 2006. DOI: `10.1785/0120060041`.

[100] D. Marcuse, C. R. Menyuk, and P. K. A. Wai. Application of the Manakov-PMD equation to studies of signal propagation in optical fibers with randomly varying birefringence. *Journal of Lightwave Technology*, 15(9):1735–1746, 1997.

[101] G. S. Martin, R. Wiley, and K. J. Marfurt. Elastic-marmousi-model. online. URL: `https://s3.amazonaws.com/open.source.geoscience/open_data/elastic-marmousi/elastic-marmousi-model.tar.gz`.

[102] G. S. Martin, R. Wiley, and K. J. Marfurt. Marmousi2: an elastic upgrade for marmousi. *The Leading Edge*, 25(2):156–166, 2006. DOI: `10.1190/1.2172306`.

[103] Y. Masson, P. Cupillard, Y. Capdeville, and B. Romanowicz. On the numerical implementation of time-reversal mirrors for tomographic imaging. *Geophysical Journal International*, 196(3):1580–1599, 2014. DOI: `10.1093/gji/ggt459`.

[104] MATLAB. *version 9.7.0 (R2019b)*. The MathWorks Inc., Natick, Massachusetts, 2020.

[105] A. Mayeli. Non-convex optimization via strongly convex majorization-minimization. *Canadian Mathematical Bulletin*, 63:1–10, 2019. DOI: `10.4153/S0008439519000730`.

[106] J. McCalpin. Trends in system cost and performance balances and implications for the future of hpc, 2015. DOI: `10.1145/2834899.2834901`.

[107] A. Mecozzi, Cr. Antonelli, and M. Shtaif. Coupled Manakov equations in multimode fibers with strongly coupled groups of modes. *Optics Express*, 20(21):23436–23441, 2012.

[108] Message Passing Interface Forum. MPI: a message-passing interface standard, 2019. URL: `https://www.mpi-forum.org/docs/drafts/mpi-2019-draft-report.pdf`.

[109] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84, 2009.

[110] P. Micikevicius. Multi-GPU Programming. online, 2011. URL: `https://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf`.

[111] P. Moczo, J. Kristek, and P. Franek. Lecture notes on rheological models. URL: `http://www.earthphysics.sk/mainpage/stud_mat/Moczo_Kristek_Franek_Rheological_Models.pdf`.

[112] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543–574, 1992. DOI: 10.1146/annurev.aa.30.090192.002551.

[113] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703–1759, 2005. DOI: 10.1088/0034-4885/68/8/r01.

[114] E. Montagne and A. Ekambaram. An optimal storage format for sparse matrices. *Information Processing Letters*, 90(2):87 –92, 2004. DOI: 10.1016/j.ipl.2004.01.014.

[115] V. Monteiller, S. Chevrot, D. Komatitsch, and Y. Wang. Three-dimensional full waveform inversion of short-period teleseismic wavefields based upon the sem–dsm hybrid method. *Geophysical Journal International*, 202:811–827, 2015.

[116] S. Mumtaz, R.-J. Essiambre, and G. P. Agrawal. Nonlinear Propagation in Multimode and Multicore Fibers: Generalization of the Manakov Equations. *Journal of Lightwave Technology*, 31(3):398–406, 2013.

[117] V. Natoli. A decade of accelerated computing augurs well for GPUs, 2019. URL: https://www.nextplatform.com/2019/07/10/a-decade-of-accelerated-computing-augurs-well-for-gpus/.

[118] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, 2010.

[119] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2nd edition, 2006.

[120] NVIDIA. CUDA C++ PROGRAMMING GUIDE, 2020. URL: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.

[121] NVIDIA. CUDA occupancy calculator. URL: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[122] NVIDIA. NVIDIA A100 tensor core gpu, 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf.

[123] NVIDIA. NVLINK-FABRIC, 2020. URL: https://www.nvidia.com/en-us/data-center/nvlink/.

[124] NVIDIA Collective Communication Library (NCCL) Documentation. online, 2018. URL: https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/index.html.

[125] NVIDIA Corp. cuFFT – NVIDIA CUDA fast Fourier transform library, 2016. URL: https://developer.nvidia.com/cufft.

[126] openacc-standard.org. OpenACC programming and best practices guide, 2015. URL: `https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf`.

[127] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013. URL: `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`.

[128] M. Osorno, M. Schirwon, N. Kijanski, R. Sivanesapillai, H. Steeb, and D. Göddeke. A cross-platform, high-performance SPH toolkit for image-based flow simulations on the pore scale of porous media. *Computer Physics Communications*, 2020.

[129] S. Pachnicke, A. Chachaj, M. Helf, and P. Krummrich. Fast parallel simulation of fiber optical communication systems accelerated by a graphics processing unit. In pages 1–4, 2010. DOI: `10.1109/ICTON.2010.5549002`.

[130] R. E. Plessix. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International*, 167(2):495–503, 2006.

[131] F. Poletti and P. Horak. Description of ultrashort pulse propagation in multimode optical fibers. *Journal of the Optical Society of America B*, 25(10):1645–1654, 2008. DOI: `10.1364/JOSAB.25.001645`.

[132] C. Ravaut, M. Alerini, S. P. Lescoffit, and E. Thomassen. Sub-salt imaging by full-waveform inversion: a parameter analysis. In 2008.

[133] D. J. Richardson, J. M. Fini, and L. E. Nelson. Space-division multiplexing in optical fibres. *Nature Photonics*, 7(5):354–362, 2013.

[134] N. Ricker. The form and laws of propagation of seismic wavelets. *Geophysics*, 18(1):10–40, 1953. DOI: `10.1190/1.1437843`.

[135] F. Rubio, M. Hanzich, A. Farrés, J. de la Puente, and J. M. Cela. Finite-difference staggered grids in gpus for anisotropic elastic wave propagation simulation. *Computers & Geosciences*, 70:181 –189, 2014. DOI: `10.1016/j.cageo.2014.06.003`.

[136] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003. DOI: `10.1137/1.9780898718003`.

[137] V. Sarkar, W. Harrod, and A. E. Snavely. Software challenges in extreme scale systems. *Journal of Physics: Conference Series*. DOI: `10.1088/1742-6596/180/1/012045`.

[138] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24(111):647–647, 1970. DOI: `10.1090/s0025-5718-1970-0274029-x`.

[139] P. Sillard. Few-Mode Fibers for Space Division Multiplexing. In *Optical Fiber Communication Conference (OFC)*, Anaheim, CA, USA, 2016. DOI: `10.1364/OFC.2016.Th1J.1`. paper Th1J.1.

[140] R. Sivanesapillai. *Pore-scale study of non-Darcian fluid flow in porous media using smoothed-particle hydrodynamics.* Doctoral thesis, Ruhr-Universität Bochum, 2016.

[141] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli. Summarizing CPU and GPU design trends with product data, 2019. arXiv: `1911.11313 [cs.DC]`.

[142] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: application to small water clusters. *The Journal of Chemical Physics*, 76(1):637–649, 1982. DOI: `/10.1063/1.442716`.

[143] T. Alkhalifah. An anisotropic Marmousi model. URL: `http://sepwww.stanford.edu/public/docs/sep95/tariq3/paper_html/index.html`.

[144] T. R. Taha and X. Xu. Parallel Split-Step Fourier Methods for the Coupled Nonlinear Schrödinger Type Equations. *Journal of Supercomputing*, 32(1):5–23, 2005. DOI: `10.1007/s11227-005-0183-5`.

[145] O. Talagrand and P. Courtier. Variational assimilation of meteorological observations with the adjoint vorticity equation. I. Theory. *Q. J. R. Meteorol. Soc.*, 113:1311–1328, 1987.

[146] C. Tape, Q. Liu, and J. Tromp. Finite-frequency tomography using adjoint methods-methodology and examples using membrane surface waves. *Geophysical Journal International*, 168(3):1105–1129, 2007. DOI: `10.1111/j.1365-246x.2006.03191.x`.

[147] A. Tarantola and B. Valette. Generalized nonlinear inverse problems solved using the least squares criterion. *Rev. Geophys. Space Phys.*, 20(2):219–232, 1982.

[148] A. N. Tikhonov and Vasiliy Yakovlevich Arsenin. Solutions of ill-posed problems. In 1977.

[149] TOP500 List - June 1997 — TOP500 Supercomputer Sites. URL: `https://www.top500.org/lists/top500/1997/06/`.

[150] TOP500 List - June 2009 — TOP500 Supercomputer Sites. URL: `https://www.top500.org/lists/top500/2009/06/`.

[151] TOP500 List - June 2019 — TOP500 Supercomputer Sites. URL: `https://www.top500.org/lists/top500/2019/06/`.

[152] TOP500 List - November 2015 — TOP500 Supercomputer Sites. URL: `https://www.top500.org/lists/top500/2015/11/`.

[153] TOP500 List - November 2020 — TOP500 Supercomputer Sites. URL: `https://www.top500.org/lists/top500/2020/11/`.

[154] Google Cloud TPU. Cloud tensor processing units (TPUs). URL: `https://cloud.google.com/tpu/docs/tpus`.

[155] STREAM Benchmark Author McCalpin Traces System Balance Trends. Tiffany trader, 2016. URL: https://www.hpcwire.com/2016/11/07/mccalpin-traces-hpc-system-balance-trends/.

[156] J. Tromp, C. Tape, and Q. Liu. Seismic tomography, adjoint methods, time reversal and banana-doughnut kernels. *Geophysical Journal International*, 160(1):195–216, 2005. DOI: 10.1111/j.1365-246X.2004.02453.x.

[157] A. Uvarov, N. Karelin, I. Koltchanov, A. Richter, H. Louchet, and G. Shkred. GPU-assisted simulations of SDM systems. In *International Conference on Transparent Optical Networks (ICTON)*, Girona, Spain, 2017. DOI: 10.1109/ICTON.2017.8025061. paper We.D1.6.

[158] M. van Driel and T. Nissen-Meyer. Optimized viscoelastic wave propagation for weakly dissipative media. *Geophysical Journal International*, 199(2):1078–1093, 2014. DOI: 10.1093/gji/ggu314.

[159] A. Venkatesh, K. Hamidouche, S. Potluri, D. Rosetti, C.-H. Chu, and D. K. Panda. MPI-GDS: High Performance MPI Designs with GPUDirect-aSync for CPU-GPU Control Flow Decoupling. In *46th International Conference on Parallel Processing (ICPP)*, pages 151–160, Bristol, UK, 2017. DOI: 10.1109/ICPP.2017.24.

[160] L. Verlet. Computer ‘experiments’ on classical fluids. I. thermodynamical properties of Lennard-Jones molecules. *Physical review*, 159(1):98, 1967.

[161] R. Versteeg. The Marmousi experience: Velocity model determination on a synthetic complex data set. en. *The Leading Edge*, 13(9):927–936, 1994. DOI: 10.1190/1.1437051.

[162] Y. Wang, S. Chevrot, V. Monteiller, D. Komatitsch, F. Mouthereau, G. Manatschal, M. Sylvander, J. Diaz, M. Ruiz, F. Grimaud, S. Benahmed, H. Pauchet, and R. Martin. The deep roots of the western pyrenees revealed by full waveform inversion of teleseismic p waves. *Geology*, 44(6):475–478, 2016. DOI: 10.1130/g37812.1.

[163] Y. Wang, J. Yang, W. Yin, and Y. Zhang. A new alternating minimization algorithm for total variation image reconstruction. *SIAM J. Imaging Sciences*, 1:248–272, 2008. DOI: 10.1137/080724265.

[164] G. H. Weiss and A. A. Maradudin. The Baker-Hausdorff Formula and a Problem in Crystal Physics. *Journal of Mathematical Physics*, 3(4):771–777, 1962.

[165] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming.* Addison-Wesley Professional, 1st edition, 2013.

[166] P. Wolfe. Convergence conditions for ascent methods. *SIAM Review*, 11(2):226–235, 1969. DOI: 10.1137/1011036.

[167] P. Wolfe. Convergence conditions for ascent methods. II: some corrections. *SIAM Review*, 13(2):185–188, 1971. DOI: 10.1137/1013035.

[168] Z. Zhang, L. Chen, and X. Bao. A fourth-order Runge-Kutta in the interaction picture method for numerically solving the coupled nonlinear Schrödinger equation. *Optics Express*, 18(8):8261–8276, 2010.

[169] S. M. Zoldi, V. Ruban, A. Zenchuk, and S. Burtsev. Parallel Implementation of the Split-step Fourier Method For Solving Nonlinear Schrödinger Systems. *SIAM News*, 32(1):1–5, 1999.

A

Appendix

A.1 Seismic waveform modeling and inversion

This section contains more detailed calculations, further tests and descriptions of transformations from chapter 5, which were omitted there to improve the reading flow.

A.1.1 Determination of the stress relaxation function

We derive the stress relaxation function, which is used in on page 109 in section 5.2.2. We use equation (5.16) to determine the relaxation:

$$\Psi(t) = \mathcal{F}^{-1}\left\{\frac{M(\omega)}{i\omega}\right\}$$

Next, we insert equation (5.43) and get

$$\Psi(t) = \mathcal{F}^{-1}\left\{M_R\frac{1+i\omega\tau_\varepsilon}{i\omega(1+i\omega\tau_\sigma)}\right\}$$

Instead of applying the inverse Fourier transform straight forward, we first rewrite equation (5.43) so that we can apply the inverse Fourier transform to simpler terms. We rewrite equation (5.43) as:

$$\begin{aligned}
\frac{M(\omega)}{i\omega} =& M_R\frac{1+i\omega\tau_\varepsilon}{i\omega(1+i\omega\tau_\sigma)} = M_R\frac{1+i\omega\tau_\varepsilon}{i\omega-\omega^2\tau_\sigma} = M_R\frac{\omega^{-1}+i\tau_\varepsilon}{i-\omega\tau_\sigma}\\
=& M_R\left(\frac{1}{i\omega-\omega^2\tau_\sigma}+\frac{i\tau_\varepsilon}{i-\omega\tau_\sigma}\right)\\
=& M_R\left(\frac{1+i\omega\tau_\sigma-i\omega\tau_\sigma}{i\omega-\omega^2\tau_\sigma}+\frac{i\tau_\varepsilon}{i-\omega\tau_\sigma}\right)\\
=& M_R\left(\frac{1+i\omega\tau_\sigma}{i\omega-\omega^2\tau_\sigma}-\frac{i\tau_\sigma}{i-\omega\tau_\sigma}+\frac{i\tau_\varepsilon}{i-\omega\tau_\sigma}\right)\\
=& M_R\left(\frac{1+i\omega\tau_\sigma}{i\omega(1+i\omega\tau_\sigma)}-\frac{i\tau_\sigma}{i-\omega\tau_\sigma}+\frac{i\tau_\varepsilon}{i-\omega\tau_\sigma}\right)\\
=& M_R\left(\frac{1}{i\omega}-\frac{i\tau_\sigma}{i-\omega\tau_\sigma}+\frac{i\tau_\varepsilon}{i-\omega\tau_\sigma}\right)
\end{aligned}$$

Next, we can apply the inverse Fourier transformation:

$$\Psi(t) = \mathcal{F}^{-1}\left\{M_r\left[\frac{-i}{\omega} + \frac{i\tau_\varepsilon}{i-\tau_\sigma\omega} - \frac{i\tau_\sigma}{i-\tau_\sigma\omega}\right]\right\}$$

To compute the inverse Fourier transform, we use the information that the function is a signal, i.e. the function is equal to zero for $t < 0$, in the time domain. Additionally, we use the following (forward) Fourier transformations:

$$\begin{aligned}
\mathcal{F}\left[a\right] &= \int_0^\infty a\,\exp(-i\omega t)\mathrm{d}t = \left[-\frac{a}{i\omega}\exp(-i\omega t)\right]_0^\infty = \frac{a}{i\omega}\\
\mathcal{F}\left[\exp(-at)\right] &= \int_0^\infty \exp(-a\,t)\exp(-i\omega t)\mathrm{d}t = \int_0^\infty \exp(-(a+i\omega)t)\mathrm{d}t\\
&= \left[-\frac{1}{a+i\omega}\exp(-(a+i\omega)t)\right]_0^\infty = \frac{1}{a+i\omega}
\end{aligned}$$

Where $a \in \mathbb{R}$ is a constant variable. Inserting the related terms leads to:

$$\begin{aligned}
\mathcal{F}^{-1}\left[\frac{-i}{\omega}\right] =& \mathcal{F}^{-1}\left[\frac{1}{i\omega}\right] = \mathcal{F}^{-1}\left[\mathcal{F}\left[1\right]\right] = 1\\
\mathcal{F}^{-1}\left[\frac{i\tau_\varepsilon}{i-\tau_\sigma\omega}\right] =& \frac{\tau_\varepsilon}{\tau_\sigma}\mathcal{F}^{-1}\left[\frac{1}{\tau_\sigma^{-1}+i\omega}\right] = \frac{\tau_\varepsilon}{\tau_\sigma}\mathcal{F}^{-1}\left[\mathcal{F}\left[\exp(-\tau_\sigma^{-1}t)\right]\right]\\
=& \frac{\tau_\varepsilon}{\tau_\sigma}\exp\left(\frac{-t}{\tau_\sigma}\right)\\
\mathcal{F}^{-1}\left[\frac{i\tau_\sigma}{i-\tau_\sigma\omega}\right] =& \mathcal{F}^{-1}\left[\frac{1}{\tau_\sigma^{-1}+i\omega}\right] = \mathcal{F}^{-1}\left[\mathcal{F}\left[\exp(-\tau_\sigma^{-1}t)\right]\right]\\
=& \exp\left(\frac{-t}{\tau_\sigma}\right)
\end{aligned}$$

All in all, we combine the three inverse Fourier transformations and get the relaxation function:

$$\begin{aligned}
\Psi(t) =& M_R\left[1 - \frac{\tau_\varepsilon}{\tau_\sigma}\exp(\frac{-t}{\tau_\sigma}) + \exp(\frac{-t}{\tau_\sigma})\right]\\
\Psi(t) =& M_R\left[1 - \left(1 - \frac{\tau_\varepsilon}{\tau_\sigma}\right)\exp(-t/\tau_\sigma)\right]
\end{aligned}$$

A.1.2 System of second order PDEs to system of first order PDEs

On page 118 in section 5.2.3, a system of second-order PDEs is transformed to a system of first-order PDEs. The detailed transformation is described below. Starting from the system of second order PDEs, we derive a system of first order PDEs. The PDEs of second order are:

$$\rho\partial_{tt}u_i = u_i\partial_j\sigma_{ij} + f_i \tag{A.1}$$

$$\sigma_{xx} = (\lambda_U + 2\mu_U)\partial_x u_x + \lambda_U\partial_y u_y + (\lambda+\mu)\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} + \mu\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} \tag{A.2}$$

$$\sigma_{yy} = \lambda_U\partial_x u_x + (\lambda_U + 2\mu_U)\partial_y u_y + (\lambda+\mu)\sum_{l=1}^{N_{\mathrm{SLS}}} R_{1,l} - \mu\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xx,l} \tag{A.3}$$

$$\sigma_{xy} = \mu_U(\partial_x u_y + \partial_y u_x) + \mu\sum_{l=1}^{N_{\mathrm{SLS}}} R_{xy,l} \tag{A.4}$$

$$\partial_t R_{1,l} = -\,R_{1,l}/\tau_{1\sigma,l} + \frac{1}{\tau_{1\sigma,l}}\left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\mathrm{tr}(\varepsilon) \tag{A.5}$$

$$\partial_t R_{xx,l} = -\,R_{xx,l}/\tau_{2\sigma,l} + \frac{1}{\tau_{2\sigma,l}}\left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)(\partial_x u_x - \partial_y u_y) \tag{A.6}$$

$$\partial_t R_{xy,l} = -\,R_{xy,l}/\tau_{2\sigma,l} + \frac{1}{\tau_{2\sigma,l}}\left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)(\partial_x u_y + \partial_y u_x) \tag{A.7}$$

We define $v = \dot{u}$ i.e. $v_x = \dot{u}_x$ and $v_y = \dot{u}_y$, substitute this in eq. A.1 and get:

$$\rho\partial_t v_x = \partial_x\sigma_{xx} + \partial_y\sigma_{yx} + f_x$$
$$\rho\partial_t v_y = \partial_x\sigma_{xy} + \partial_y\sigma_{yy} + f_y$$

Differentiating equation A.2-A.4 and substituting $v = \dot{u}$ gives us

$$\partial_t\sigma_{xx} = (\lambda_U + 2\mu_U)\partial_x v_x + \lambda_U\partial_y v_y + (\lambda+\mu)\sum_{l=1}^{N_{\mathrm{SLS}}} \bar{R}_{1,l} + \mu\sum_{l=1}^{N_{\mathrm{SLS}}} \bar{R}_{xx,l}$$

$$\partial_t\sigma_{yy} = \lambda_U\partial_x v_x + (\lambda_U + 2\mu_U)\partial_y v_y + (\lambda+\mu)\sum_{l=1}^{N_{\mathrm{SLS}}} \bar{R}_{1,l} - \mu\sum_{l=1}^{N_{\mathrm{SLS}}} \bar{R}_{xx,l}$$

$$\partial_t\sigma_{xy} = \mu_U(\partial_x v_y + \partial_y v_x) + \mu\sum_{l=1}^{N_{\mathrm{SLS}}} \bar{R}_{xy,l}$$

with

$$
\begin{aligned}
\bar{R}^l_1 &= \partial_t R^l_1 = \partial_t \left(\frac{1}{\tau_{1\sigma}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \mathrm{tr}(\varepsilon)(\cdot) \right) \right) \\
&= \frac{1}{\tau_{1\sigma}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \mathrm{tr}(\dot{\varepsilon})(\cdot) \right) \\
\bar{R}^l_{xx} &= \partial_t R^l_{xx} = \partial_t \left(\frac{1}{\tau_{2\sigma}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{2\sigma,l}}) * \mathrm{tr}(\partial_x u_x - \partial_y u_y))(\cdot) \right) \right) \\
&= \frac{1}{\tau_{2\sigma}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{2\sigma,l}}) * \mathrm{tr}(\partial_x v_x - \partial_y v_y))(\cdot) \right) \\
\bar{R}^l_{xy} &= \partial_t R^l_{xy} = \partial_t \left(\frac{1}{\tau_{2\sigma}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{2\sigma,l}}) * \mathrm{tr}(\partial_x u_y + \partial_y u_x)(\cdot) \right) \right) \\
&= \frac{1}{\tau_{2\sigma}} \left(1 - \frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{2\sigma,l}}) * \mathrm{tr}(\partial_x v_y + \partial_y v_x)(\cdot) \right)
\end{aligned}
$$

Here, we first used the rule for convolutions that $\partial_t(f*g) = (\partial_t f)*g = f*\partial_t g$, followed by inserting the substitution $v = \dot{u}$. Similar to the first order PDEs, we obtain the differential equations for the memory variables by differentiating the corresponding equations, which leads to the following equations:

$$
\begin{aligned}
\bar{R}^l_1 =& \frac{1}{\tau_{1\sigma}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \mathrm{tr}(\dot{\varepsilon})(\cdot) \right) \\
\partial_t \bar{R}^l_1 =& \frac{1}{\tau_{1\sigma}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \partial_t \left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \mathrm{tr}(\dot{\varepsilon})(\cdot) \right) \\
=& \frac{1}{\tau_{1\sigma}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \partial_t \mathrm{tr}(\dot{\varepsilon})(\cdot) \right) \\
=& \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \left[\exp(-(t-t')/\tau_{1\sigma,l}) \mathrm{tr}(\dot{\varepsilon}) \right]^t_{-\infty} \\
&- \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \int_{-\infty}^t \frac{1}{\tau_{1\sigma,l}} \exp(-(t-t')/\tau_{1\sigma,l}) \mathrm{tr}(\dot{\varepsilon}) dt' \\
=& \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \left[\exp(0) \mathrm{tr}(\dot{\varepsilon}) - 0 \right] \\
&- \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right) \underbrace{\int_{-\infty}^t \frac{1}{\tau_{1\sigma,l}} \exp(-(t-t')/\tau_{1\sigma,l}) \mathrm{tr}(\dot{\varepsilon}) dt'}_{= \frac{1}{\tau_{1\sigma,l}} \left(\exp(-\frac{\cdot}{\tau_{1\sigma,l}}) * \mathrm{tr}(\dot{\varepsilon})(\cdot) \right)} \\
=& -\bar{R}^l_1 / \tau_{1\sigma,l} + \mathrm{tr}(\dot{\varepsilon}) \frac{1}{\tau_{1\sigma,l}} \left(1 - \frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)
\end{aligned}
$$

All in all we get the system of first order PDEs

$$\begin{aligned}
\rho\partial_t v_x =&\partial_x\sigma_{xx} + \partial_y\sigma_{xy} + f_x\\
\rho\partial_t v_y =&\partial_x\sigma_{xy} + \partial_y\sigma_{yy} + f_y\\
\partial_t\sigma_{xx} =&(\lambda+2\mu)\partial_x v_x + \lambda\partial_y v_y + (\mu+\lambda)\sum_{l=1}^{N_{\mathrm{SLS}}}\bar{R}_{1,l} + \mu\sum_{l=1}^{N_{\mathrm{SLS}}}\bar{R}_{xx,l}\\
\partial_t\sigma_{yy} =&\lambda\partial_x v_x + (\lambda+2\mu)\partial_y v_y + (\mu+\lambda)\sum_{l=1}^{N_{\mathrm{SLS}}}\bar{R}_{1,l} - \mu\sum_{l=1}^{N_{\mathrm{SLS}}}\bar{R}_{xx,l}\\
\partial_t\sigma_{xy} =&\mu(\partial_x v_y + \partial_y v_y) + \mu\sum_{l=1}^{N_{\mathrm{SLS}}}\bar{R}_{xy,l}\\
\partial_t\bar{R}_{1,l} =&-\bar{R}_{1,l}/\tau_{1\sigma,l} + \frac{1}{\tau_{1\sigma,l}}\left(1-\frac{\tau_{1\varepsilon,l}}{\tau_{1\sigma,l}}\right)\mathrm{tr}(\dot{\varepsilon})\\
\partial_t\bar{R}_{xx,l} =&-\bar{R}_{xx,l}/\tau_{2\sigma,l} + \frac{1}{\tau_{2\sigma,l}}\left(1-\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)(\partial_x v_x - \partial_y v_y)\\
\partial_t\bar{R}_{xy,l} =&-\bar{R}_{xy,l}/\tau_{2\sigma,l} + \frac{1}{\tau_{2\sigma,l}}\left(1-\frac{\tau_{2\varepsilon,l}}{\tau_{2\sigma,l}}\right)(\partial_x v_y + \partial_y v_x)
\end{aligned}$$

A.1.3 Relaxation and modulus function for the modified forward model

The transfer from the standard Zener approach to the approach that allows the Q factors to be inverted is performed in section 5.3.3 on page 127. We describe the detailed substitution in the following.
We recall the modulus function (5.47) and the relaxation function (5.48) for the standard approach, which are:

$$\begin{aligned}
M(\omega) =& \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1 + i\omega\tau_{\varepsilon,l}}{1 + i\omega\tau_{\sigma,l}} \\
\Psi(t) =& M_R \left[1 - \frac{1}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \left(1 - \frac{\tau_{\varepsilon,l}}{\tau_{\sigma,l}} \right) \exp(-t/\tau_{\sigma,l}) \right] H(t)
\end{aligned}$$

Next we insert the definitions $\tau_{\varepsilon,l} = \frac{1+N_{\text{SLS}}d_l}{\theta_l}$, $\tau_{\sigma,l} = \frac{1}{\theta_l}$, for the modulus function these result in the following:

$$\begin{aligned}
M(\omega) =& \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1 + i\omega\tau_{\varepsilon,l}}{1 + i\omega\tau_{\sigma,l}} \\
=& \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{1 + i\omega\frac{1+N_{\text{SLS}}d_l}{\theta_l}}{1 + i\omega\frac{1}{\theta_l}} \\
=& \frac{M_R}{N_{\text{SLS}}} \sum_{l=1}^{N_{\text{SLS}}} \frac{\theta_l + i\omega}{\theta_l + i\omega} \frac{i\omega N_{\text{SLS}} d_l}{\theta_l + i\omega} \\
=& M_R \left(1 + \sum_{l=1}^{N_{\text{SLS}}} \frac{i\omega d_l(\theta_l - i\omega)}{\theta_l^2 + \omega^2} \right)
\end{aligned}$$

Inserting the definition $d_l := Q^{-1}D_l$ yields

$$= M_R \left(1 + \frac{1}{Q} \sum_{l=1}^{N_{\text{SLS}}} \frac{i\omega D_l(\theta_l - i\omega)}{\theta_l^2 + \omega^2} \right).$$

Similar, we insert the definitions $\tau_{\varepsilon,l} = \frac{1+N_{\mathrm{SLS}}d_l}{\theta_l}$, $\tau_{\sigma,l} = \frac{1}{\theta_l}$, for the relaxation function these result in the following:

$$
\begin{aligned}
\Psi(t) =& M_R \left[1 - \frac{1}{N_{\mathrm{SLS}}} \sum_{l=1}^{N_{\mathrm{SLS}}} \left(1 - \frac{\tau_{\varepsilon,l}}{\tau_{\sigma,l}}\right) \exp(-t/\tau_{\sigma,l})\right] H(t) \\
=& M_R \left[1 - \frac{1}{N_{\mathrm{SLS}}} \sum_{l=1}^{N_{\mathrm{SLS}}} \left(1 - \frac{1+N_{\mathrm{SLS}}d_l}{\theta_l}\theta_l\right) \exp(-t\theta_l)\right] H(t) \\
=& M_R \left[1 - \frac{1}{N_{\mathrm{SLS}}} \sum_{l=1}^{N_{\mathrm{SLS}}} (1 - 1 - N_{\mathrm{SLS}}d_l) \exp(-t\theta_l)\right] H(t) \\
=& M_R \left[1 + \sum_{l=1}^{N_{\mathrm{SLS}}} d_l \exp(-t\theta_l)\right] H(t)
\end{aligned}
$$

Inserting the definition $d_l := Q^{-1}D_l$ yields

$$
= M_R \left[1 + \frac{1}{Q} \sum_{l=1}^{N_{\mathrm{SLS}}} (D_l) \exp(-t\theta_l)\right] H(t).
$$

A.1.4 Derivative of the misfit function

The calculation of the derivative of the misfit function, which was not performed in section 5.4.5, is calculated in detail in the following.

After we have derived the adjoint problem, we can now use equation (5.88) to determine the derivative of the misfit function.

First we calculate the derivative for the linear operator of the forward problem

$$\mathbf{L}(u,m) = -A^{-1}\partial_t u + Du + A^{-1}Bu + A^{-1}s,$$

which is

$$\begin{aligned}\frac{d\mathbf{L}(u,m)}{\mathrm{d}m} &= -\partial_m A^{-1}\partial_t u + \partial_m\left(A^{-1}B\right)u + \partial_m A^{-1}s\\ &= -\partial_m A^{-1}\partial_t u + \partial_m A^{-1}Bu + A^{-1}\partial_m Bu + \partial_m A^{-1}s\\ &= \partial_m A^{-1}\underbrace{(-\partial_t u + Bu + s)}_{=-ADu} + A^{-1}\partial_m Bu\\ &= -\partial_m A^{-1}(AD)u + A^{-1}\partial_m Bu\end{aligned}$$

Next, we can insert this derivative into equation (5.88) and obtain

$$\frac{\mathrm{d}\chi}{\mathrm{d}m}\delta m = \int\int u^* \cdot \left(-\partial_m A^{-1}(AD) + A^{-1}\partial_m B\right)u\ \mathrm{d}x\mathrm{d}t. \tag{A.8}$$

Now we are able to calculate the derivative. First, we calculate all needed matrices and the necessary partial derivatives of the matrices. The inverse of matrix A reads

$$A^{-1} = \begin{pmatrix} \frac{\lambda+2\mu}{4\mu(\lambda+\mu)} & -\frac{\lambda}{4\mu(\lambda+\mu)} & 0 & 0 & 0 & 0 & 0 & 0\\ -\frac{\lambda}{4\mu(\lambda+\mu)} & \frac{\lambda+2\mu}{4\mu(\lambda+\mu)} & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & \frac{1}{\mu} & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & \rho & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & \rho & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{\theta_{1,l}D_{1,l}} & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\theta_{2,l}D_{2,l}} & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\theta_{2,l}D_{2,l}} \end{pmatrix}.$$

For the sake of completeness, we also specify the matrix product AD, which is

$$AD = \begin{pmatrix} 0 & 0 & 0 & (\lambda+2\mu)\partial_x & \lambda\partial_y & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda\partial_x & (\lambda+2\mu)\partial_y & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu\partial_y & \mu\partial_x & 0 & 0 & 0 \\ \frac{\partial_x}{\rho} & 0 & \frac{\partial_y}{\rho} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial_y}{\rho} & \frac{\partial_x}{\rho} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -D_{1,l}\theta_{1,l}\partial_x & -D_{1,l}\theta_{1,l}\partial_y & 0 & 0 & 0 \\ 0 & 0 & 0 & -D_{2,l}\theta_{2,l}\partial_x & D_{2,l}\theta_{2,l}\partial_y & 0 & 0 & 0 \\ 0 & 0 & 0 & -D_{2,l}\theta_{2,l}\partial_y & -D_{2,l}\theta_{2,l}\partial_x & 0 & 0 & 0 \end{pmatrix}.$$

To facilitate the readability in the following, we calculate ADu and introduce new names for the result

$$(AD)u = \begin{pmatrix} (\lambda+2\mu)\partial_x v_x + \lambda\partial_y v_y \\ \lambda\partial_x v_x + (\lambda+2\mu)\partial_y v_y \\ \mu\partial_y v_x + \mu\partial_x v_y \\ \frac{\partial_x}{\rho}\sigma_{xx} + \frac{\partial_y}{\rho}\sigma_{xy} \\ \frac{\partial_y}{\rho}\sigma_{yy} + \frac{\partial_x}{\rho}\sigma_{xy} \\ -D_{1,l}\theta_{1,l}\partial_x v_x - D_{1,l}\theta_{1,l}\partial_y v_y \\ -D_{2,l}\theta_{2,l}\partial_x v_x + D_{2,l}\theta_{2,l}\partial_y v_y \\ -D_{2,l}\theta_{2,l}\partial_y v_x - D_{2,l}\theta_{2,l}\partial_x v_y \end{pmatrix} =: \begin{pmatrix} \partial_t\tilde{\sigma}_{xx} \\ \partial_t\tilde{\sigma}_{yy} \\ \partial_t\tilde{\sigma}_{xy} \\ \partial_t\tilde{v}_x \\ \partial_t\tilde{v}_y \\ \partial_t R_{1,l} \\ \partial_t R_{xx,l} \\ \partial_t R_{xy,l} \end{pmatrix}$$

Next, we calculate all partial derivatives of matrix B. B contains the parameters λ, μ, Q_1 and Q_2, so we need to determine the partial derivatives with respect to these parameters.

These result in:

$$\partial_\lambda B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & \frac{1}{Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\partial_\mu B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & \frac{1}{Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l}} & \frac{1}{Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l}} & -\frac{1}{Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\partial_{Q_1} B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -(\lambda+\mu)\frac{1}{(Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l})^2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -(\lambda+\mu)\frac{1}{(Q_1+\sum_{l=1}^{N_{\text{SLS}}} D_{1,l})^2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\partial_{Q_2} B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -\mu\frac{1}{(Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l})^2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mu\frac{1}{(Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l})^2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu\frac{1}{(Q_2+\sum_{l=1}^{N_{\text{SLS}}} D_{2,l})^2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Before we can determine the partial derivatives of the misfit function, we need the partial derivatives of the matrix A^{-1}. This matrix contains the parameters ρ, λ and μ, so we need to determine the partial derivatives in direction of these parameters. They are given by:

$$\partial_\rho A^{-1} = \begin{pmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$\partial_\lambda A^{-1} = \begin{pmatrix} -\frac{1}{4(\lambda+\mu)^2} & -\frac{1}{4(\lambda+\mu)^2} & 0&0&0&0&0&0\\ -\frac{1}{4(\lambda+\mu)^2} & -\frac{1}{4(\lambda+\mu)^2} & 0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$\partial_\mu A^{-1} = \begin{pmatrix} -\frac{\lambda^2+2\lambda\mu+2\mu^2}{4\mu^2(\lambda+\mu)^2} & \frac{\lambda(\lambda+2\mu)}{4\mu^2(\lambda+\mu)^2} & 0&0&0&0&0&0\\ \frac{\lambda(\lambda+2\mu)}{4\mu^2(\lambda+\mu)^2} & -\frac{\lambda^2+2\lambda\mu+2\mu^2}{4\mu^2(\lambda+\mu)^2} & 0&0&0&0&0&0\\ 0&0&-\frac{1}{\mu^2}&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

Finally, we calculate the partial derivatives of the misfit function using equation (A.8). The partial derivative in ρ direction is simplified because only A is dependent on ρ. This results in

$$\begin{aligned} u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}\rho} &= u^* \cdot (-\partial_\rho A^{-1}(AD) + A^{-1}\partial_\rho B)u \\ &= u^* \cdot (-\partial_\rho A^{-1}(AD))u \\ &= -v_x^*\partial_t\tilde{v}_x - v_y^*\partial_t\tilde{v}_y. \end{aligned}$$

Since μ is contained in both A and B, the partial derivatives of both matrices appear, and the partial derivative in μ direction of the misfit function is

$$\begin{aligned} u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}\lambda} &= u^* \cdot (-\partial_\lambda A^{-1}(AD) + A^{-1}\partial_\lambda B)u \\ &= \frac{1}{4(\lambda+\mu)^2}\left\{(\sigma^*_{xx} + \sigma^*_{yy})(\partial_t\tilde{\sigma}_{xx} + \partial_t\tilde{\sigma}_{yy})\right\} \\ &\quad + (\sigma^*_{xx} + \sigma^*_{yy})\frac{1}{2(\lambda+\mu)}\frac{1}{Q_1 + \sum_{l=1}^{N_{\text{SLS}}} D_{1,l}}\sum_{l=1}^{N_{\text{SLS}}} R_{1,l}. \end{aligned}$$

Also for λ the partial derivatives for A and B occur, so the partial derivative of the misfit function is

$$\begin{aligned} u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}\mu} =& u^* \cdot (-\partial_\mu A^{-1}(AD) + A^{-1}\partial_\mu B)u \\ =& \sigma^*_{xx}\left(\frac{\lambda^2 + 2\lambda\mu + 2\mu^2}{4\mu^2(\lambda+\mu)^2}\partial_t\tilde{\sigma}_{xx} - \frac{\lambda(\lambda+2\mu)}{4\mu^2(\lambda+\mu)^2}\partial_t\tilde{\sigma}_{yy}\right) \\ &- \sigma^*_{yy}\left(\frac{\lambda(\lambda+2\mu)}{4\mu^2(\lambda+\mu)^2}\partial_t\tilde{\sigma}_{xx} - \frac{\lambda^2 + 2\lambda\mu + 2\mu^2}{4\mu^2(\lambda+\mu)^2}\partial_t\tilde{\sigma}_{yy}\right) + \sigma^*_{xy}\left(\frac{1}{\mu^2}\partial_t\tilde{\sigma}_{xy}\right) \\ &+ (\sigma^*_{xx} + \sigma^*_{yy})\frac{1}{2(\lambda+\mu)}\frac{1}{Q_1 + \sum_{l=1}^{N_{\text{SLS}}} D_{1,l}}\sum_{l=1}^{N_{\text{SLS}}} R_{1,l} \\ &+ (\sigma^*_{xx} - \sigma^*_{yy})\frac{1}{2(\mu)}\frac{1}{Q_2 + \sum_{l=1}^{N_{\text{SLS}}} D_{2,l}}\sum_{l=1}^{N_{\text{SLS}}} R_{xx,l} \\ &+ (\sigma^*_{xy})\frac{1}{\mu}\frac{1}{Q_2 + \sum_{l=1}^{N_{\text{SLS}}} D_{2,l}}\sum_{l=1}^{N_{\text{SLS}}} R_{xy,l}. \end{aligned}$$

Since the quality factors are only included in the matrix B, the calculation of the partial derivative of the misfit function with respect to Q_1 and Q_2 is simplified, which results for Q_1 in

$$\begin{aligned} u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}Q_1} =& u^* \cdot (-\partial_{Q_1} A^{-1}(AD) + A^{-1}\partial_{Q_1} B)u \\ =& u^* \cdot (A^{-1}\partial_{Q_1} B)u \\ =& -\frac{1}{2}\frac{1}{\left(Q_1 + \sum_{l=1}^{N_{\text{SLS}}} D_{1,l}\right)^2}\left(\sigma^*_{xx}\sum_{l=1}^{N_{\text{SLS}}} R_{1,l} + \sigma^*_{yy}\sum_{l=1}^{N_{\text{SLS}}} R_{1,l}\right) \end{aligned}$$

and for Q_2 the following partial derivative results

$$\begin{aligned} u^* \cdot \frac{d\mathbf{L}(u,m)}{\mathrm{d}Q_2} =& u^* \cdot (-\partial_{Q_2} A^{-1}(AD) + A^{-1}\partial_{Q_2} B)u \\ =& u^* \cdot (A^{-1}\partial_{Q_2} B)u \\ =& \frac{1}{2}\frac{1}{\left(Q_2 + \sum_{l=1}^{N_{\text{SLS}}} D_{2,l}\right)^2}\left(\sigma^*_{yy}\sum_{l=1}^{N_{\text{SLS}}} R_{xx,l} - \sigma^*_{xx}\sum_{l=1}^{N_{\text{SLS}}} R_{xx,l} - 2\sigma^*_{xy}\sum_{l=1}^{N_{\text{SLS}}} R_{xy,l}\right) \end{aligned}$$

All in all, with the forward and adjoint equations and the partial derivatives of the misfit function, we now have everything we need to apply an optimization algorithm to the minimization problem (5.80).

A.1.5 Quality comparison and improvement of the Q factor approximation

In section 5.3.3, the modified Zener model approach is improved in terms of accuracy. Then, the presented improvement is numerically investigated for different Q intervals and different number of parallel Zener models N_{SLS} on page 126. In addition to the test found in section 5.3.3, the test is repeated here for other intervals and another number of Zener models N_{SLS}. The respective intervals can be found in the following paragraph headings. The findings of these tests are analogous to those in section 5.3.3.

It can be seen that this piece-wise linear correction leads to much smaller ℓ_2 and maximum errors, especially for small Q factors. For small Q factors, the model resulting from the full objective function remains better. However, if we look at the inverse Q factors Q_{max} we can see that the modified model resulting from the simplified objective function, after improvement with the shift, approximates the Q factor better than the model resulting from the full objective function after correction.

Interval $Q \in [15,\ 500]$; number of Zener models $N_{\mathbf{SLS}} = 3$

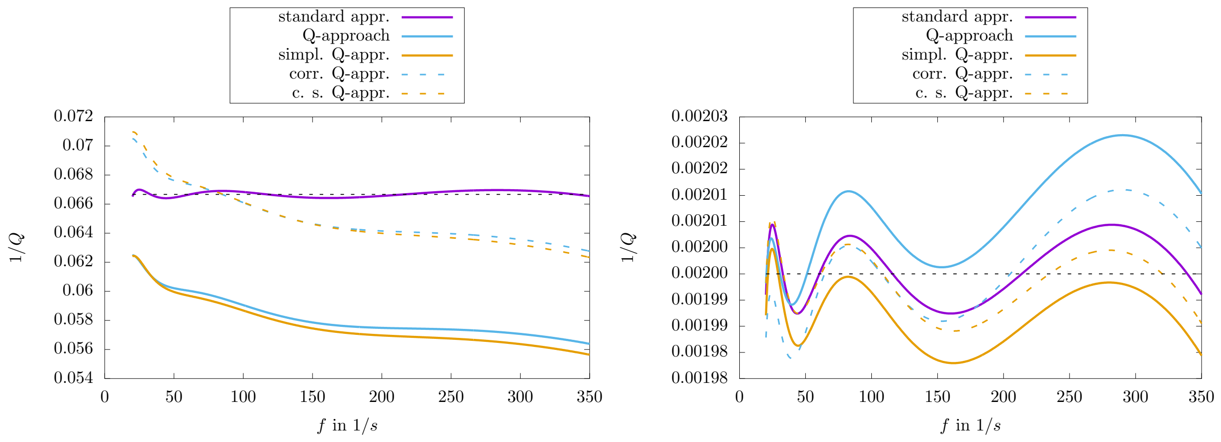


Figure A.1.1: Approximation of $Q_{\mathrm{min}} = 15$ (left) and $Q_{\mathrm{max}} = 500$ for the generalized Zener model and the modified Zener model. 'standard appr.' describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.67) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

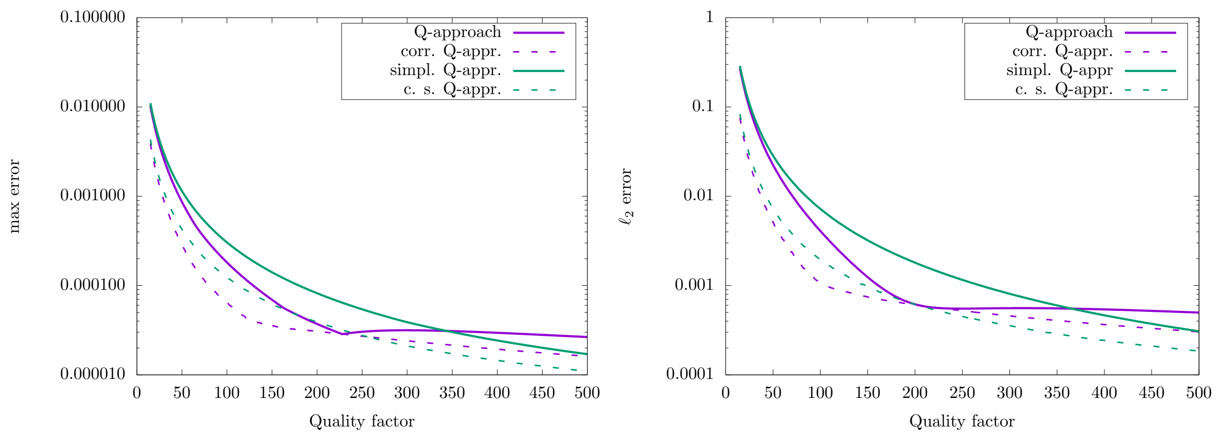


Figure A.1.2: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min},\ Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

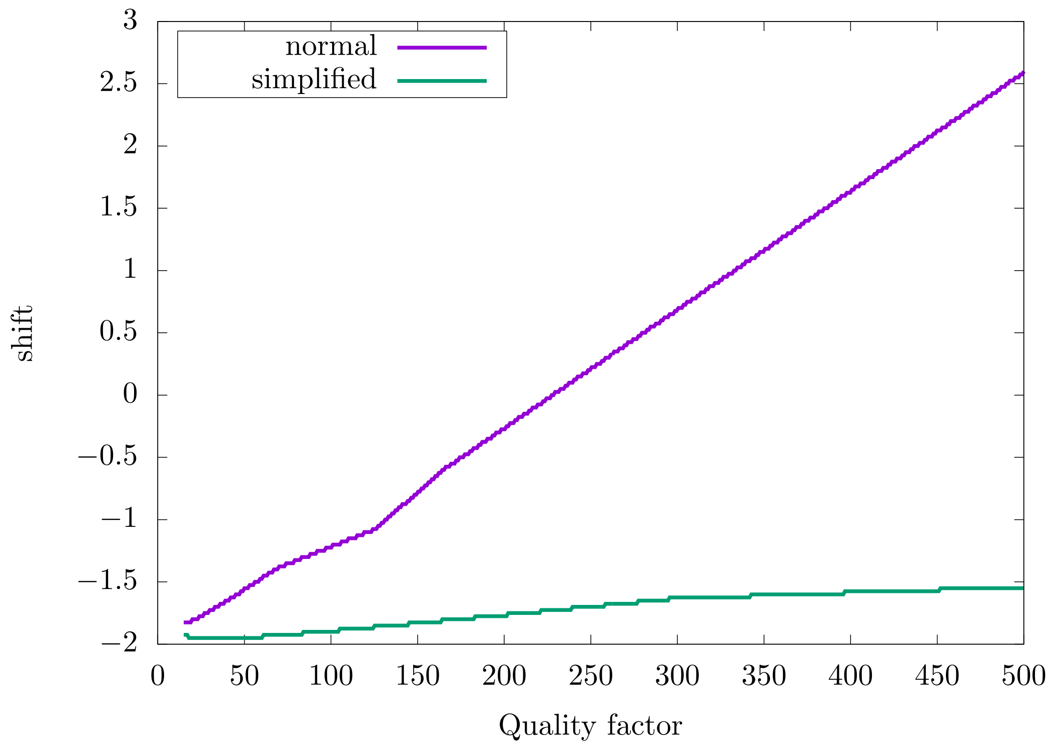


Figure A.1.3: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [15,\ 900]$; number of Zener models $N_{\mathbf{SLS}} = 3$

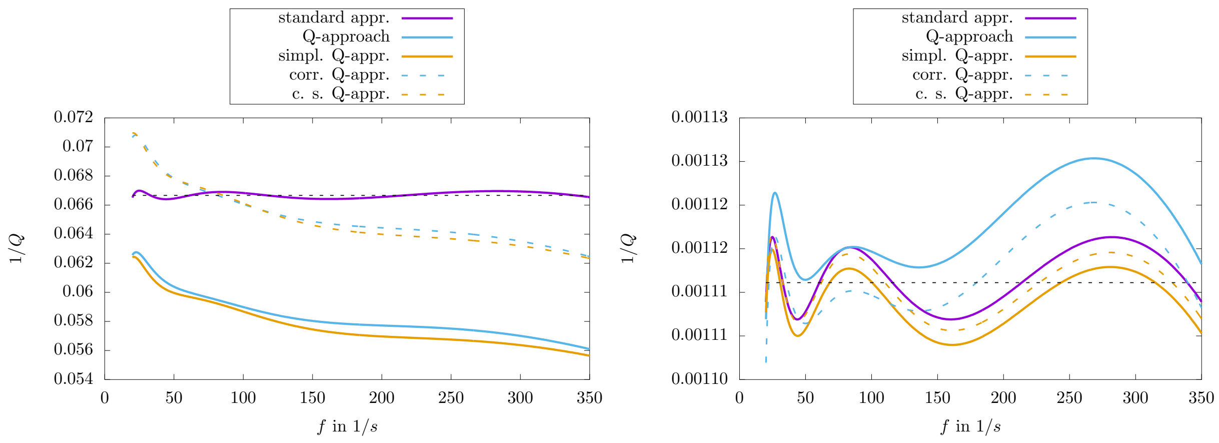


Figure A.1.4: Approximation of $Q_{\text{min}} = 15$ (left) and $Q_{\text{max}} = 900$ for the generalized Zener model and the modified Zener model. 'standard appr.' describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.67) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

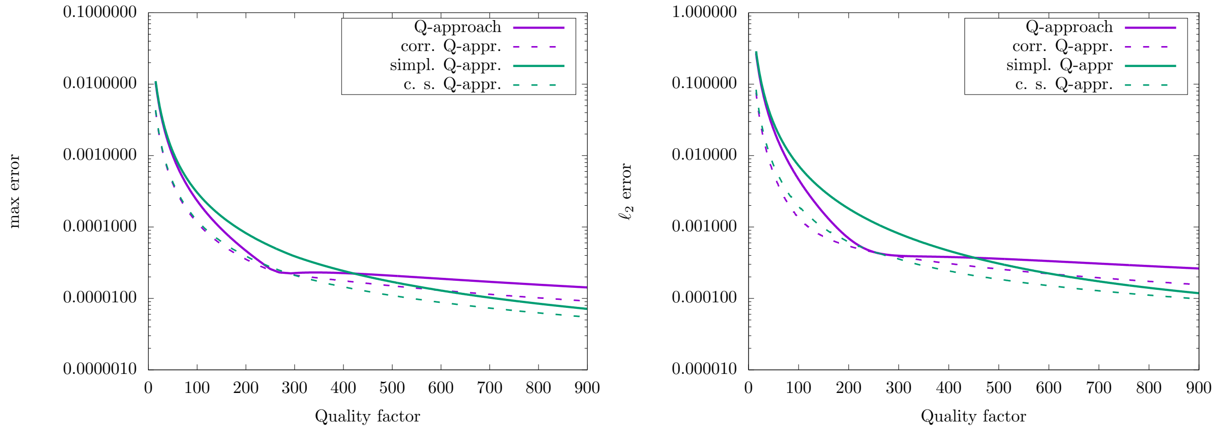


Figure A.1.5: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min},\ Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

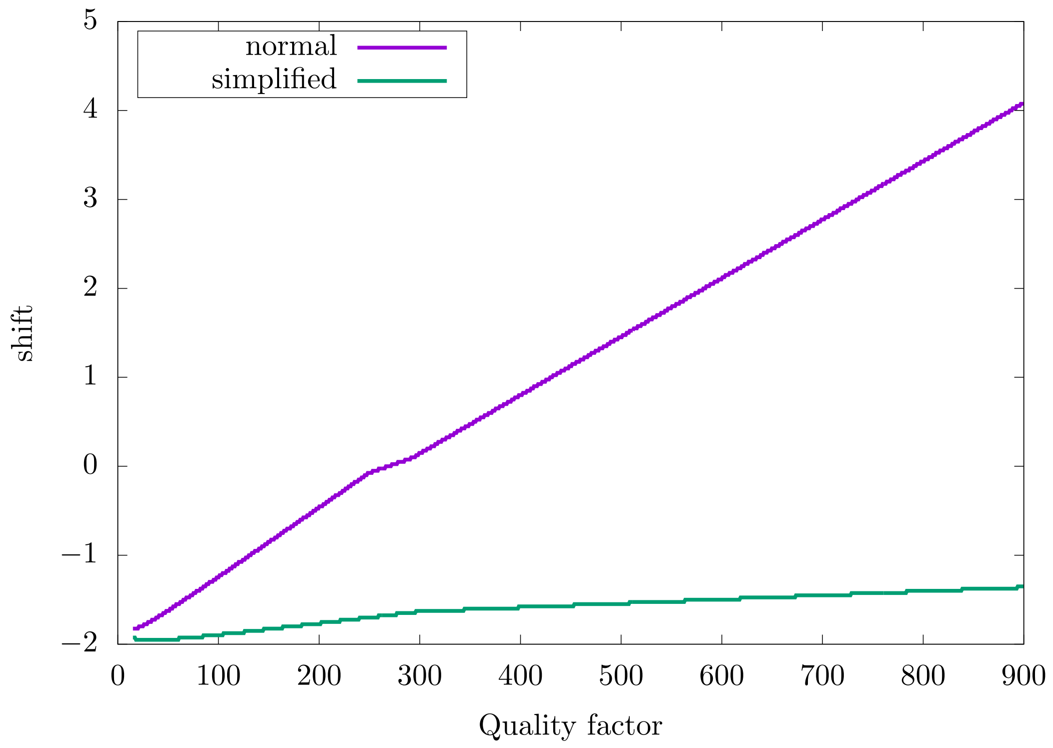


Figure A.1.6: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [800,\ 900]$; number of Zener models $N_{\mathbf{SLS}} = 3$

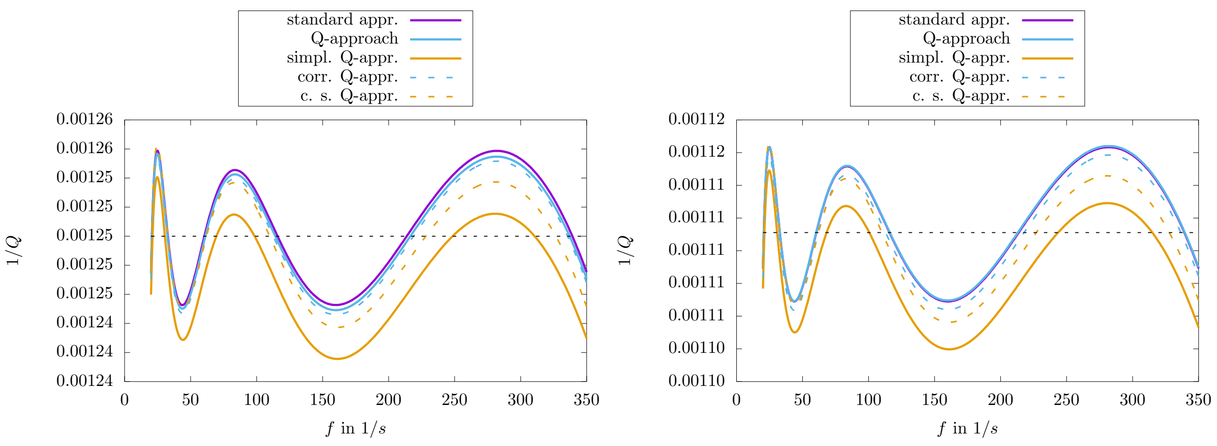


Figure A.1.7: Approximation of $Q_{\text{min}} = 800$ (left) and $Q_{\text{max}} = 900$ for the generalized Zener model and the modified Zener model. 'standard appr.' describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.67) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

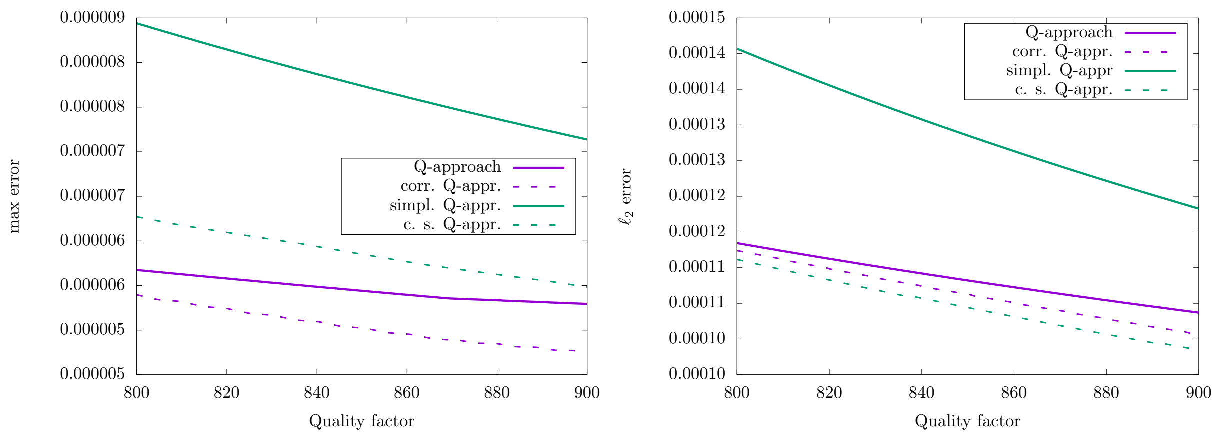


Figure A.1.8: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min},\ Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

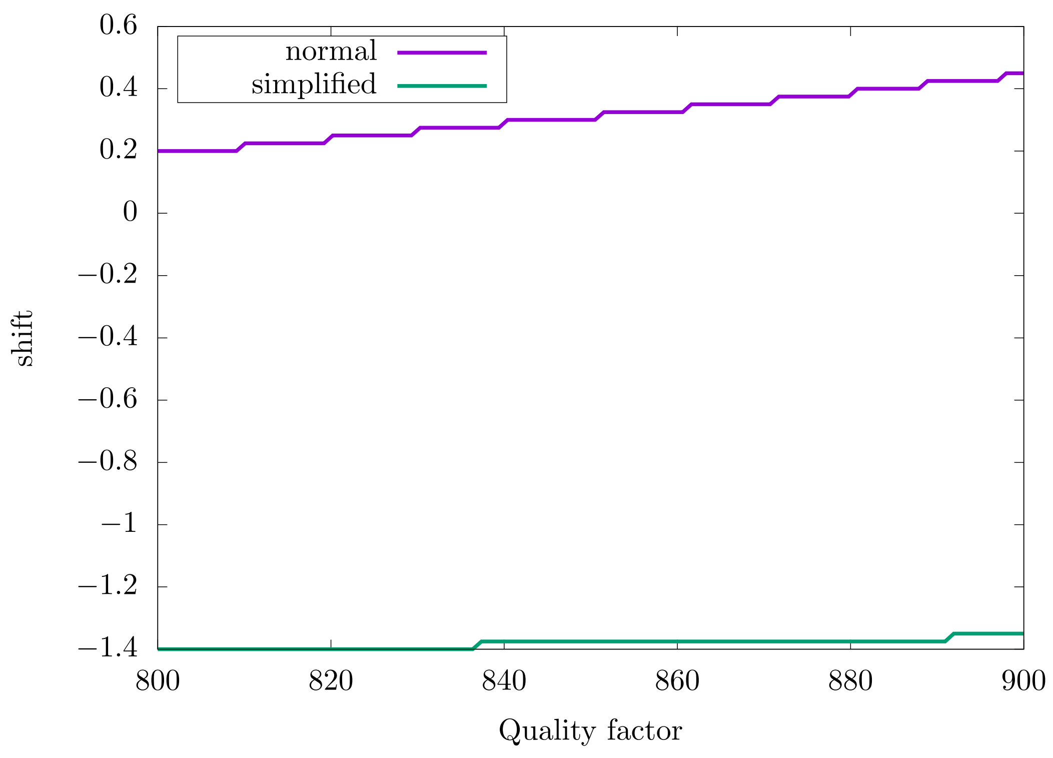


Figure A.1.9: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [500,\ 1000]$; number of Zener models $N_{\mathbf{SLS}} = 3$

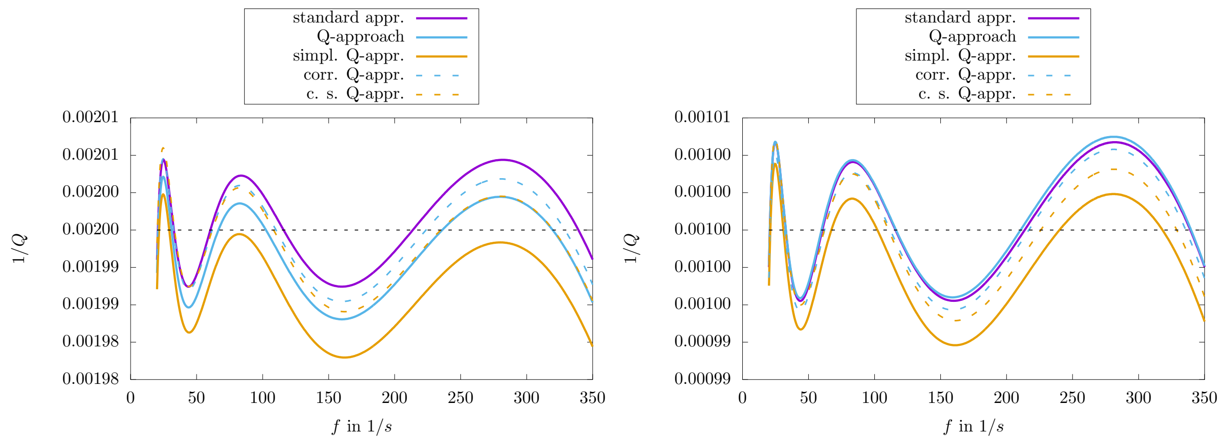


Figure A.1.10: Approximation of $Q_{\text{min}} = 500$ (left) and $Q_{\text{max}} = 1000$ for the generalized Zener model and the modified Zener model. 'standard appr.' describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.67) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

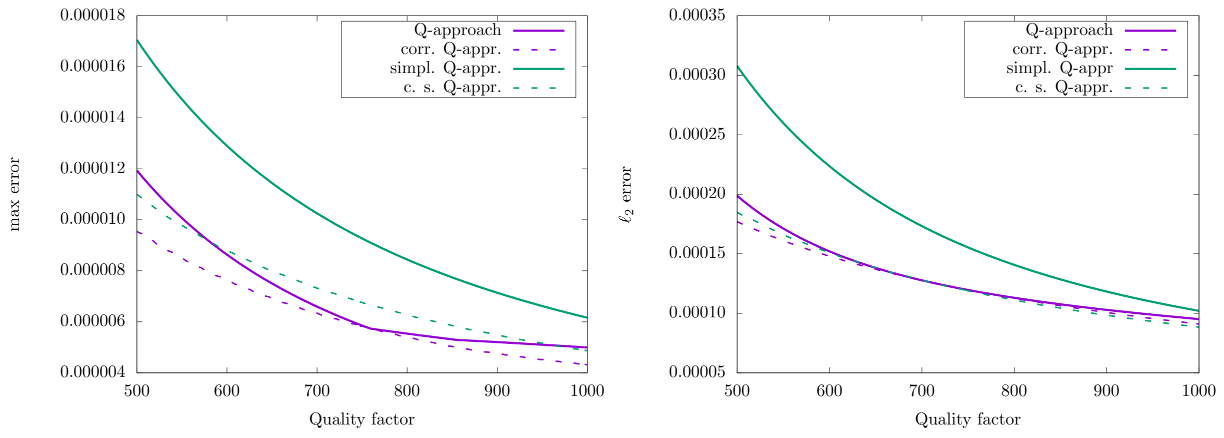


Figure A.1.11: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min},\ Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

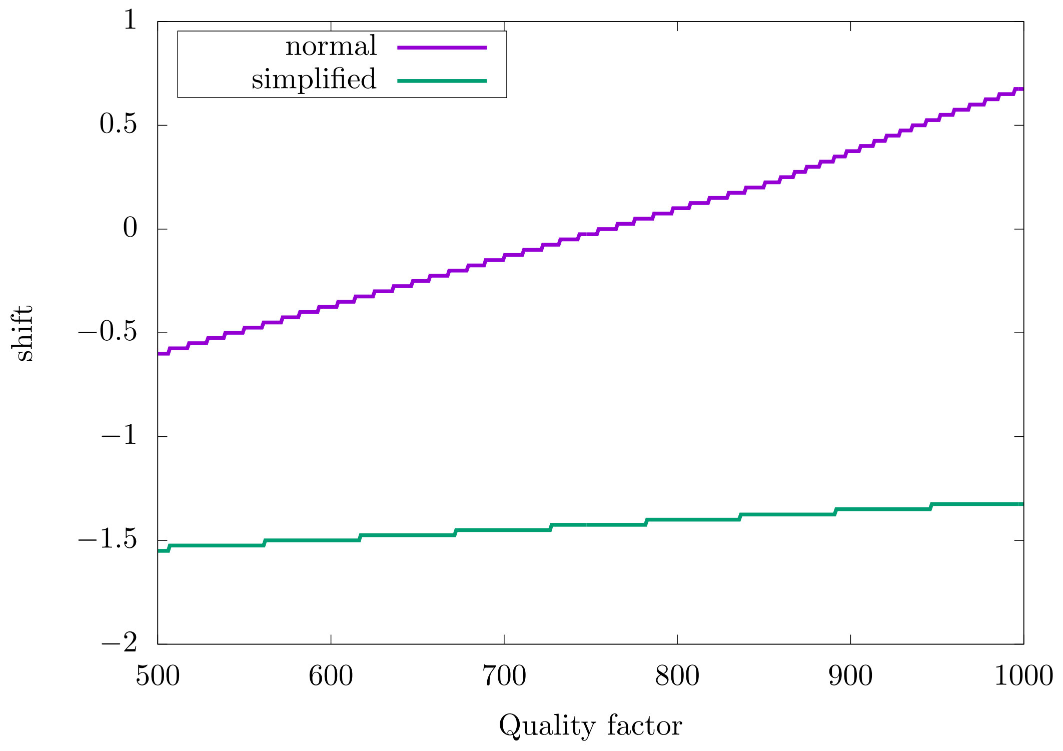


Figure A.1.12: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [15,\ 100]$; number of Zener models $N_{\mathbf{SLS}} = 3$

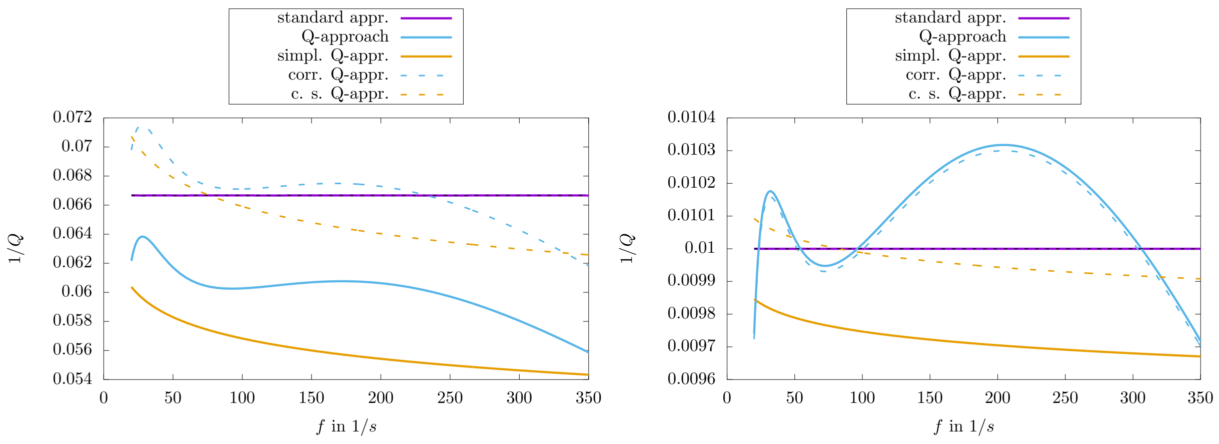


Figure A.1.13: Approximation of $Q_{\text{min}} = 15$ (left) and $Q_{\text{max}} = 100$ for the generalized Zener model and the modified Zener model. 'standard appr.' describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.67) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

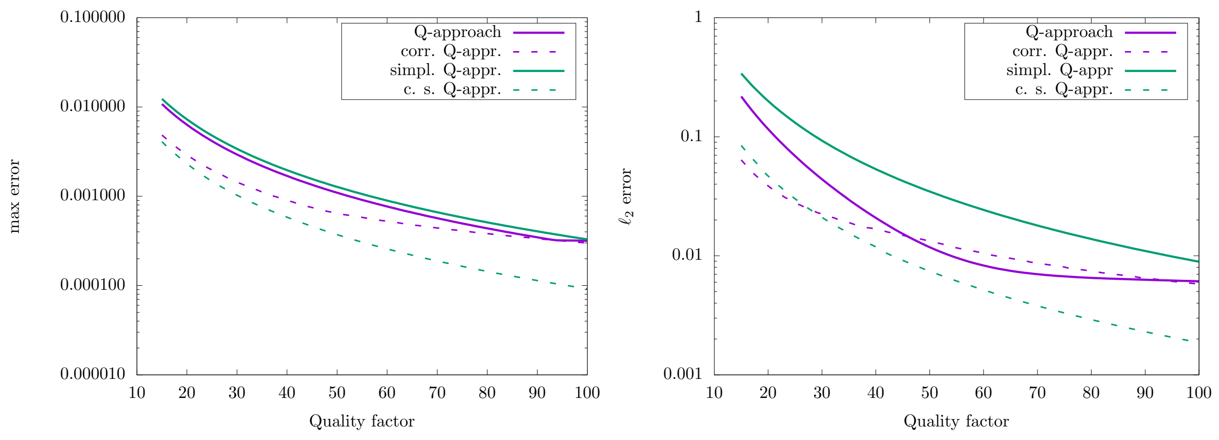


Figure A.1.14: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min},\ Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

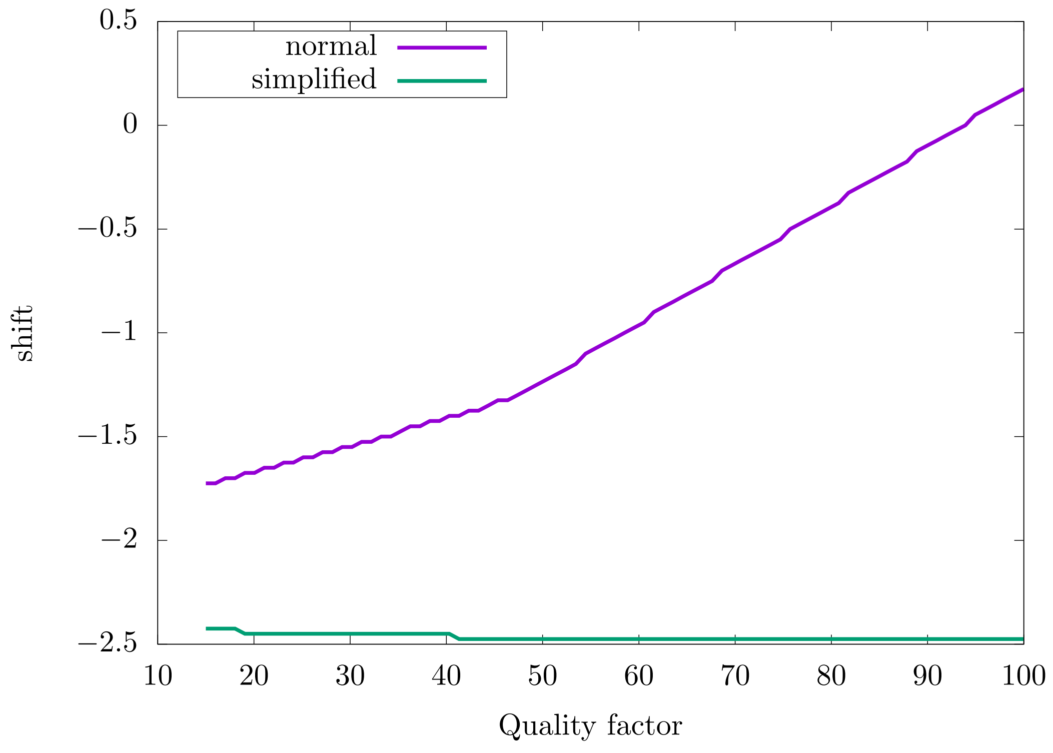


Figure A.1.15: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [15,\ 500]$; number of Zener models $N_{\mathbf{SLS}} = 10$

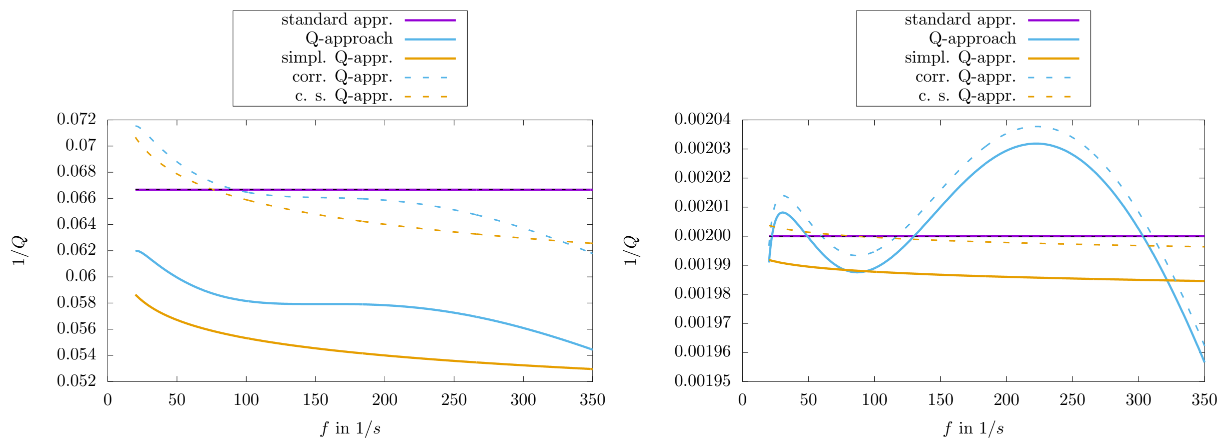


Figure A.1.16: Approximation of $Q_{\text{min}} = 15$ (left) and $Q_{\text{max}} = 500$ for the generalized Zener model and the modified Zener model. ‘standard appr.’ describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. ‘Q-approach’ names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. ‘simpl. Q-appr.’ describes the same as ‘Q-approach’ but using eq. (5.67) to obtain the relaxation parameters. ‘corr. Q-appr.’ describes the piece-wise linear correction (see sec. 5.3.3) of ‘Q-approach’. ‘c. s. Q-appr.’ describes the piece-wise linear correction (see sec. 5.3.3) of ‘simpl. Q-appr.’.

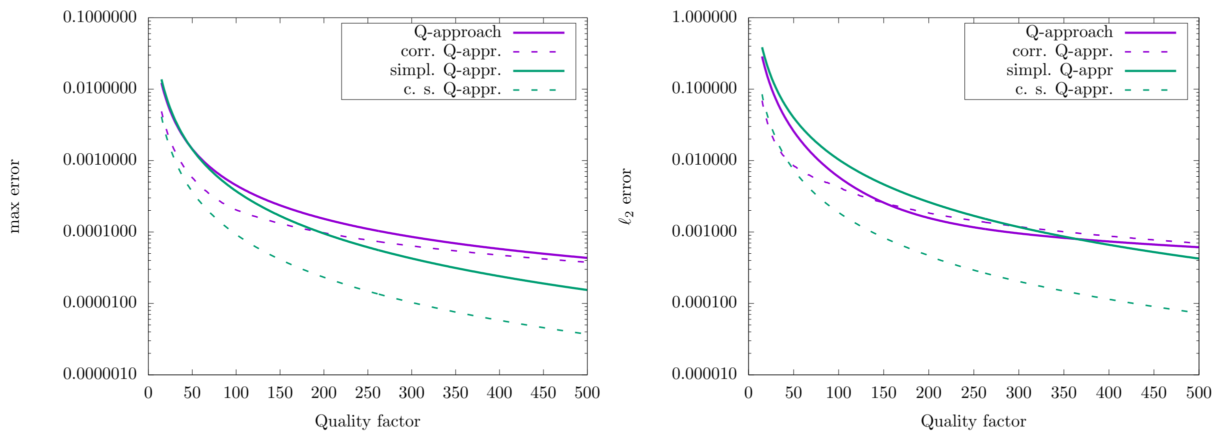


Figure A.1.17: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min}, Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

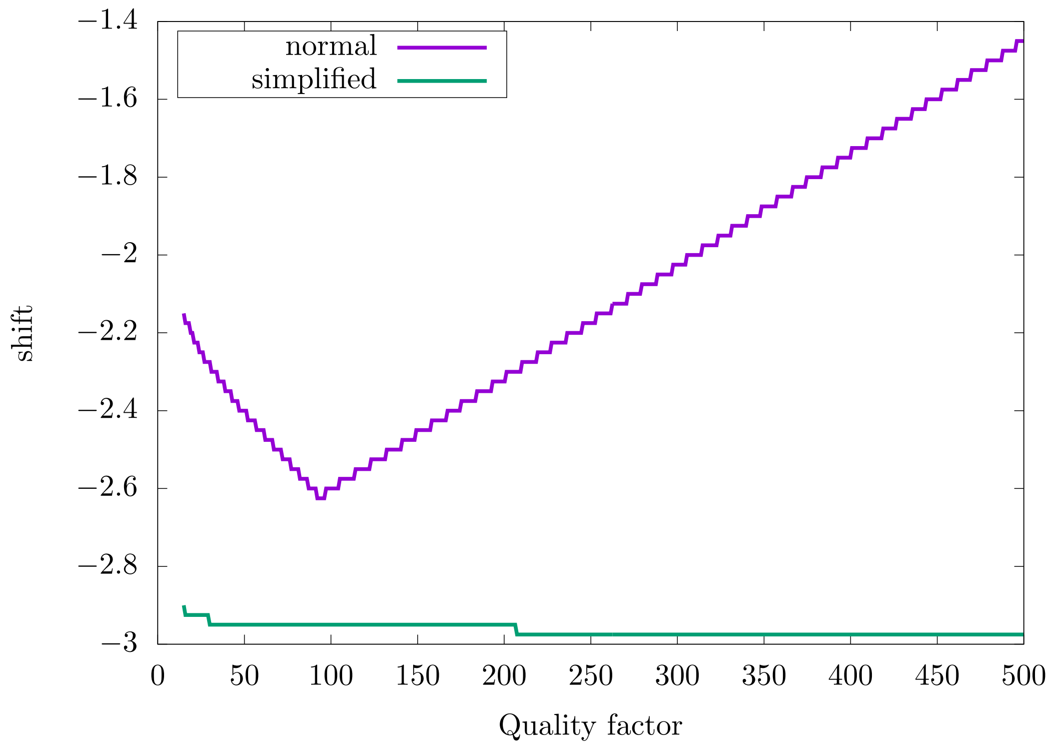


Figure A.1.18: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [15,\ 900]$; number of Zener models $N_{\mathbf{SLS}} = 10$

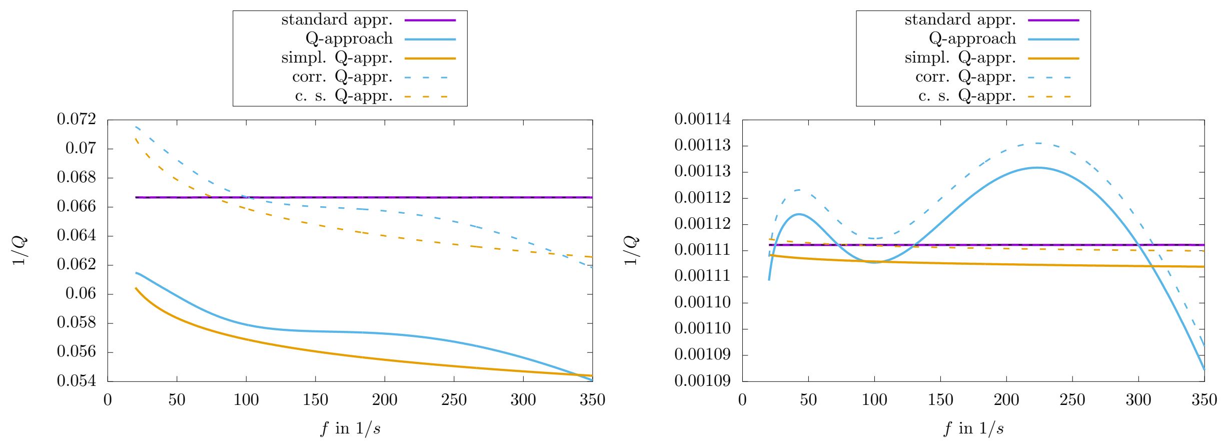


Figure A.1.19: Approximation of $Q_{\min} = 15$ (left) and $Q_{\max} = 900$ for the generalized Zener model and the modified Zener model. 'standard appr.' describes the standard generalized Zener model approach applied to $Q_{\min}$ or $Q_{\max}$. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.67) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

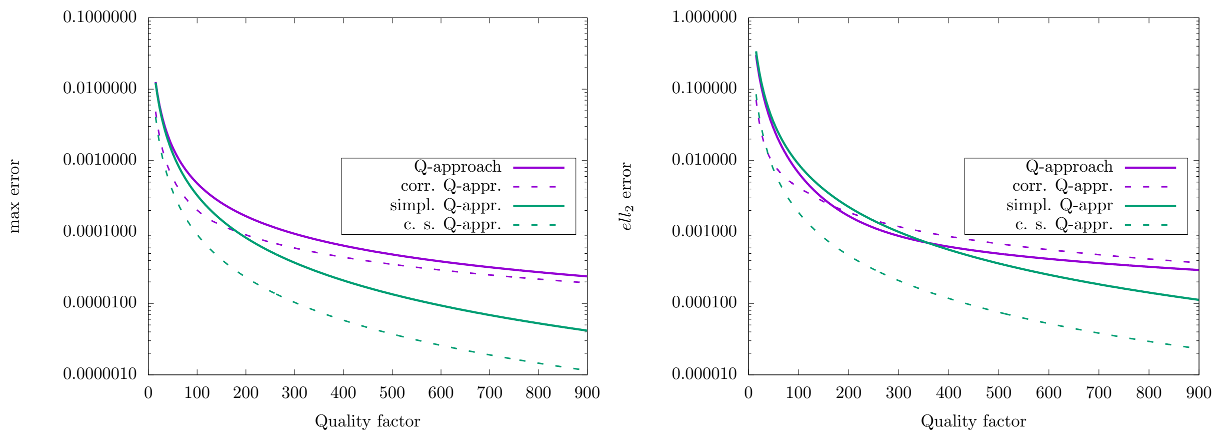


Figure A.1.20: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min},\ Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

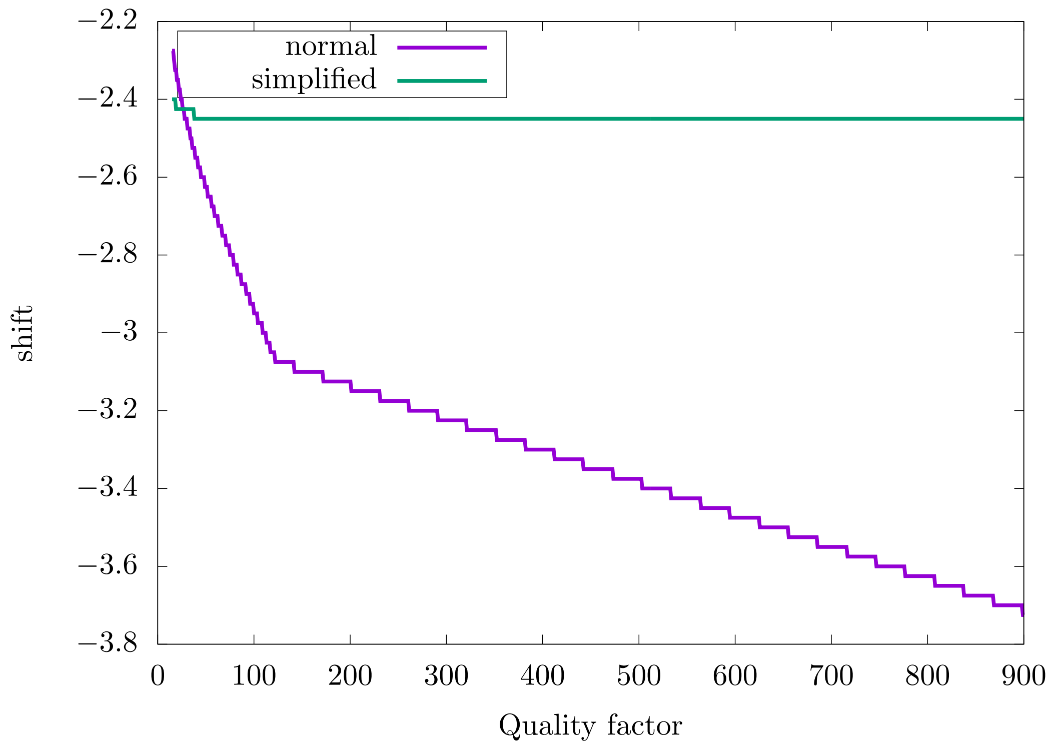
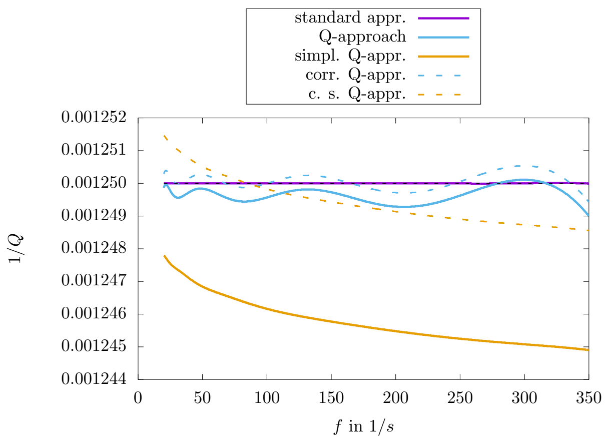


Figure A.1.21: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [800,\ 900]$; number of Zener models $N_{\mathbf{SLS}} = 10$



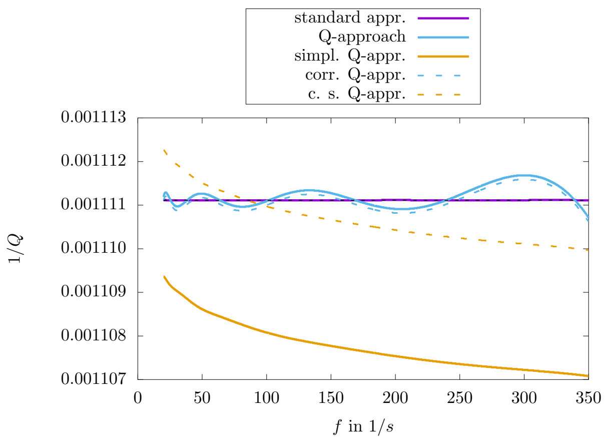


Figure A.1.22: Approximation of $Q_{\text{min}} = 800$ (left) and $Q_{\text{max}} = 900$ for the generalized Zener model and the modified Zener model. ‘standard appr.’ describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. ‘Q-approach’ names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. ‘simpl. Q-appr.’ describes the same as ‘Q-approach’ but using eq. (5.67) to obtain the relaxation parameters. ‘corr. Q-appr.’ describes the piece-wise linear correction (see sec. 5.3.3) of ‘Q-approach’. ‘c. s. Q-appr.’ describes the piece-wise linear correction (see sec. 5.3.3) of ‘simpl. Q-appr.’.

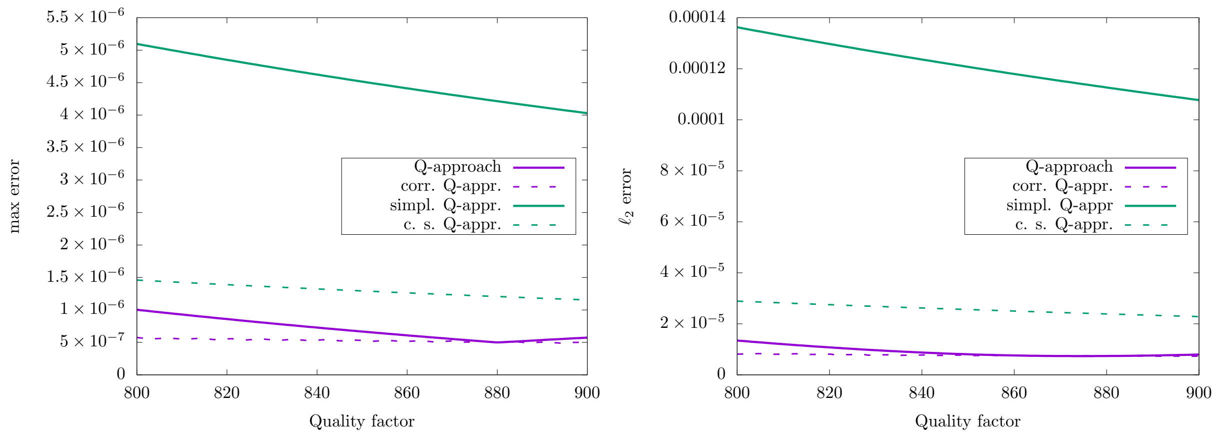


Figure A.1.23: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min},\ Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

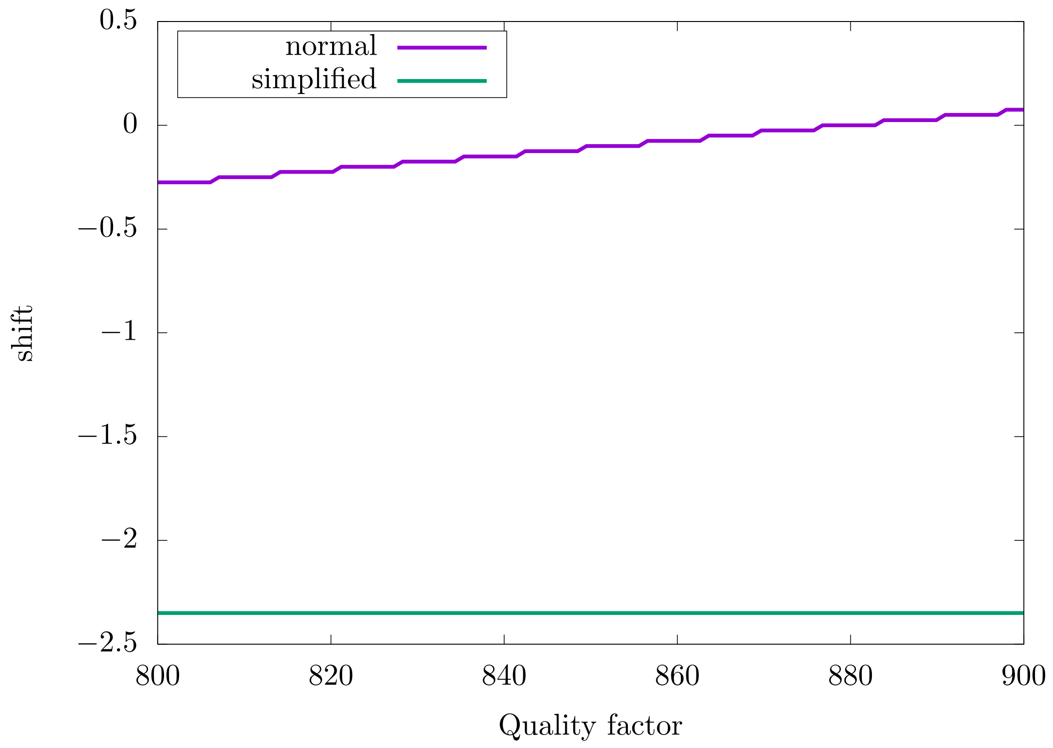


Figure A.1.24: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

Interval $Q \in [500,\ 1000]$; number of Zener models $N_{\mathbf{SLS}} = 10$

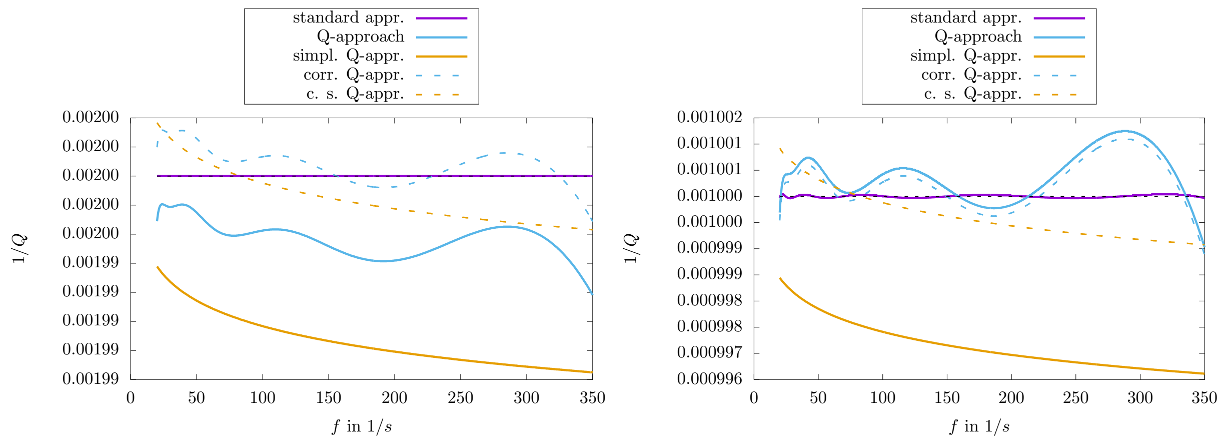


Figure A.1.25: Approximation of $Q_{\text{min}} = 500$ (left) and $Q_{\text{max}} = 1000$ for the generalized Zener model and the modified Zener model. 'standard appr.' describes the standard generalized Zener model approach applied to Q_{min} or Q_{max}. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.67) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

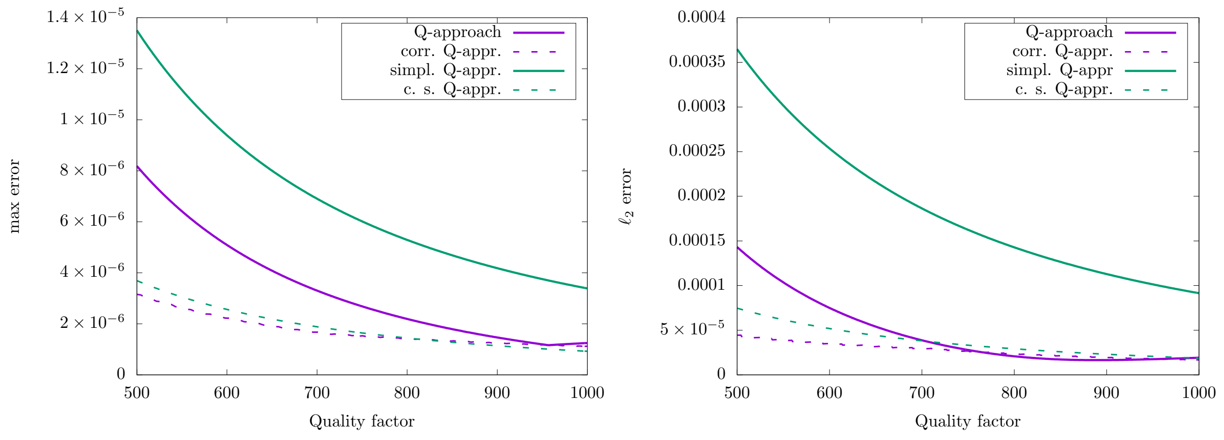


Figure A.1.26: Maximum error (left) and ℓ_2 error (right) for the approximation of $Q \in [Q_{\min}, Q_{\max}]$ for the modified Zener model. 'Q-approach' names the new approach described in section 5.3.1 using eq. (5.67) to obtain the relaxation parameters. 'simpl. Q-appr.' describes the same as 'Q-approach' but using eq. (5.68) to obtain the relaxation parameters. 'corr. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'Q-approach'. 'c. s. Q-appr.' describes the piece-wise linear correction (see sec. 5.3.3) of 'simpl. Q-appr.'.

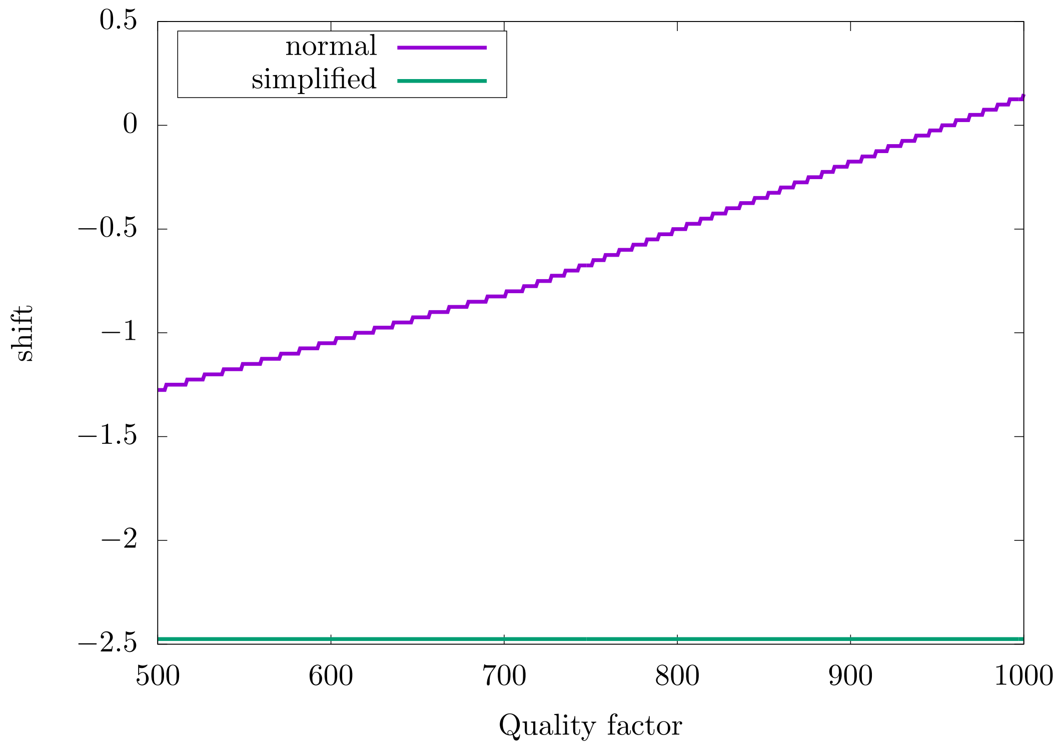


Figure A.1.27: Optimal shift parameter for the modified Zener approach described in 5.3.1 for the complete objective function (5.67) and the simplified objective function (5.67)

A.1.6 TV and total generalized p-variation regularization using the split-Bregman method

In section 5.5.1, we describe that the TV and total generalized p-variation is implemented using an alternating algorithm and the optimization problem with the variations norm is solved using the split-Bergman algorithm. An algorithmic implementation is described in the following two sections.

p-variation regularization

Using the total generalized p-variation regularization leads to the following optimization problem:

$$m^* = \min_m \left\{ \frac{1}{2} \|u(m) - u_0\|_2^2 + \lambda_{\mathcal{T}_p} \mathcal{T}_p(m) \right\}$$

We introduce an auxiliary model v and rewrite the above optimization problem into the following constrained optimization problem

$$\begin{aligned} m^* =& \min_m \left\{ \frac{1}{2} \|u(m) - u_0\|_2^2 + \lambda_{\mathcal{T}_p} \mathcal{T}_p(v) \right\}, \\ \text{s. t. } m =& v. \end{aligned}$$

Using the alternating direction method of multipliers (ADMM) procedure [30] leads to following three equations:

$$\begin{aligned} m^{(k+1)} =& \min_m \left\{ \frac{1}{2} \|u(m) - u_0\|_2^2 + \lambda_{\mathcal{T}_p,1} \|m - v^{(k)} + q^{(k)}\|_2^2 \right\} \\ v^{(k+1)} =& \min_v \left\{ \lambda_{\mathcal{T}_p,1} \|m^{(k+1)} - v + q^{(k)}\|_2^2 + \lambda_{\mathcal{T}_p,2} \mathcal{T}_p(v) \right\} \\ q^{(k+1)} =& q^{(k+1)} + m^{(k+1)} - v^{(k+1)} \end{aligned}$$

The superscript (k) refers to the k-th iteration of the FWI algorithm and q is the ADMM variable. We define $g = m^{(k+1)} + q^{(k)}$, set $\mu := \frac{\lambda_{\mathcal{T}_p,1}}{2\lambda_{\mathcal{T}_p,2}}$, insert these into

$$v^{(k+1)} = \min_v \left\{ \lambda_{\mathcal{T}_p,1} \|m^{(k+1)} - v + q^{(k)}\|_2^2 + \lambda_{\mathcal{T}_p,2} \mathcal{T}_p(v) \right\}$$

and get the following constrained optimization problem

$$\begin{aligned} \{v^{(k+1)}, w^{(k+1)}, h^{(k+1)}, s^{(k+1)}\} =& \min_{v,w,h,s} \left\{ \frac{\mu}{2} \|g - v\|_2^2 + \alpha_0 \|h\|_p^p + \alpha_1 \|s\|_p^p \right\} \\ \text{s. t. } h =& (h_x,\ h_y)^\top = \nabla v - w \\ s =& \begin{pmatrix} s_{xx} & s_{xy} \\ s_{xy} & s_{yy} \end{pmatrix} = \boldsymbol{\varepsilon}(w) \end{aligned}$$

Using the split-Bregman iteration technique [63], we rewrite the constrained optimization problem as the following optimization system:

$$\begin{aligned}\{v^{(k+1)}, w^{(k+1)}, h^{(k+1)}, s^{(k+1)}\} = \min_{v,w,h,s} &\Big\{\alpha_0\|h\|_p^p + \frac{\eta_0}{2}\|h - \tilde{h}^{(k)} - (\nabla v - w)\|_2^2 + \alpha_1\|s\|_p^p \\ &+\frac{\eta_1}{2}\|s - \tilde{s}^{(k)} - \boldsymbol{\varepsilon}(w)\|_2^2 + \frac{\mu}{2}\|g^{(k+1)} - v\|_2^2\Big\} \\ \tilde{h}^{(k+1)} =&\tilde{h}^{(k)} + \left[(\nabla v^{(k+1)} - w^{(k+1)}) - h^{(k+1)}\right] \\ \tilde{s}^{(k+1)} =&\tilde{s}^{(k)} + \left[\boldsymbol{\varepsilon}(w^{(k+1)} - s^{(k+1)}\right]\end{aligned}$$

$\tilde{h}$ and $\tilde{s}$ are the so-called split-Bregman dual variables. We decompose this optimization system into the following four subproblems:

$$\begin{aligned} v^{(k+1)} &= \min_v \Big\{\frac{\mu}{2}\|v - g^{(k+1)}\|_2^2 + \frac{\eta_0}{2}\|h^{(k)} - \tilde{h}^{(k)} - (\nabla v - w^{(k)})\|_2^2\Big\} \\ w^{(k+1)} &= \min_w \Big\{\frac{\eta_0}{2}\|h^{(k)} - \tilde{h}^{(k)} - (\nabla v^{(k+1)} - w)\|_2^2 + \frac{\eta_1}{2}\|s^{(k)} - \tilde{s}^{(k)} - \boldsymbol{\varepsilon}(w)\|_2^2\Big\} \\ h^{(k+1)} &= \min_h \Big\{\alpha_0\|h\|_p^p + \frac{\eta_0}{2}\|h - \tilde{h}^{(k)} - (\nabla v^{(k+1)} - w^{(k+1)})\|_2^2\Big\} \\ s^{(k+1)} &= \min_s \Big\{\alpha_a\|s\|_p^p + \frac{\eta_1}{2}\|s - \tilde{s}^{(k)} - \boldsymbol{\varepsilon}(w^{(k+1)})\|_2^2\Big\}\end{aligned}$$

The first-order optimality condition of the first subproblem

$$\min_v \Big\{\frac{\mu}{2}\|v - g^{(k+1)}\|_2^2 + \frac{\eta_0}{2}\|h^{(k)} - \tilde{h}^{(k)} - (\nabla v - w^{(k)})\|_2^2\Big\}$$

is:

$$(\mu I - \eta_0\nabla^\top\nabla)v^{(k+1)} = \mu g^{(k+1)} + \eta_0\nabla_x^\top(h_x^{(k)} + w_x^{(k)} - \tilde{h}_x^{(k)}) + \eta_0\nabla_y^\top(h_y^{(k)} + w_y^{(k)} - \tilde{h}_y^{(k)})$$

Using a Gauss-Seidel iteration yields

$$\begin{aligned} v_{i.j}^{(k+1)} =& \frac{\eta_0}{\mu + 4\eta_0}\left[v_{i+1,j}^{(k)} + v_{i-1,j}^{(k)} + v_{i,j+1}^{(k)} + v_{i,j-1}^{(k)}\right] \\ &+ \frac{\eta_0}{\mu + 4\eta_0}\left[h_{x|i-1,j}^{(k)} - h_{x|i,j}^{(k)} + w_{x|i-1,j}^{(k)} - w_{x|i,j}^{(k)} - (\tilde{h}_{x|i-1,j}^{(k)} - \tilde{h}_{x|i-1,j}^{(k)})\right] \\ &+ \frac{\eta_0}{\mu + 4\eta_0}\left[h_{y|i-1,j}^{(k)} - h_{y|i,j}^{(k)} + w_{y|i-1,j}^{(k)} - w_{y|i,j}^{(k)} - (\tilde{h}_{y|i-1,j}^{(k)} - \tilde{h}_{y|i-1,j}^{(k)})\right] \\ &+ \frac{\mu}{\mu + 4\eta_0} g_{i,j}^{(k+1)}.\end{aligned}$$

The first-order optimality condition of the second subproblem

$$\min_w \Big\{\frac{\eta_0}{2}\|h^{(k)} - \tilde{h}^{(k)} - (\nabla v^{(k+1)} - w)\|_2^2 + \frac{\eta_1}{2}\|s^{(k)} - \tilde{s}^{(k)} - \boldsymbol{\varepsilon}(w^{(k)})\|_2^2\Big\}$$

leads to two linear systems:

$$\begin{aligned}\left(\eta_0 I - \eta_1 \nabla_x^\top \nabla_x - \frac{1}{2}\eta_1 \nabla_y^\top \nabla_y\right) w_x^{(k+1)} =& -\eta_0(h_x^{(k)} - \tilde{h}_x^{(k)} - \nabla_x v^{(k+1)}) + \eta_1 \nabla_x^\top (s_{xx}^{(k)} - \tilde{s}_{xx}^{(k)})\\ &+ \eta_1 \nabla_y^\top (s_{xy}^{(k)} - \tilde{s}_{xy}^{(k)} - \frac{1}{2}\nabla_x w_y^{(k)})\\ \left(\eta_0 I - \frac{1}{2}\eta_1 \nabla_x^\top \nabla_x - \eta_1 \nabla_y^\top \nabla_y\right) w_y^{(k+1)} =& -\eta_0(h_y^{(k)} - \tilde{h}_y^{(k)} - \nabla_y v^{(k+1)}) + \eta_1 \nabla_y^\top (s_{yy}^{(k)} - \tilde{s}_{yy}^{(k)})\\ &+ \eta_1 \nabla_x^\top (s_{xy}^{(k)} - \tilde{s}_{xy}^{(k)} - \frac{1}{2}\nabla_y w_x^{(k)})\end{aligned}$$

We define $v_{y,x|i,j}^{(k)} = w_{y|i+1,j}^{(k)} - w_{y|i,j}^{(k)}$, use a Gauss-Seidel iteration to solve these systems and obtain:

$$\begin{aligned}w_{x|i,j}^{(k+1)} =& \frac{\eta_1}{\eta_0 + 3\eta_1}\left[w_{i+1,j}^{(k)} + w_{i-1,j}^{(k)} + \frac{1}{2}(w_{x|i,j+1}^{(k)} + w_{i,j-1}^{(k)})\right]\\ &- \frac{\eta_0}{\eta_0 + 3\eta_1}\left[h_{x|i-1,j}^{(k)} - h_{x|i,j}^{(k)} - (v_{i-1,j}^{(k)} - v_{i,j}^{(k)})\right]\\ &+ \frac{\eta_1}{\eta_0 + 3\eta_1}\left[s_{xx|i-1,j}^{(k)} - s_{xx|i,j}^{(k)} - (\tilde{s}_{xx|i-1,j}^{(k)} - \tilde{s}_{xx|i,j}^{(k)})\right]\\ &+ \frac{\eta_1}{\eta_0 + 3\eta_1}\left[s_{xy|i-1,j}^{(k)} - s_{xy|i,j}^{(k)} - (\tilde{s}_{xy|i-1,j}^{(k)} - \tilde{s}_{xy|i,j}^{(k)}) - \frac{1}{2}(v_{x,y|i,j-1}^{(k)} - v_{x,y|i,j}^{(k)})\right]\end{aligned}$$

We use the generalized p-shrinkage [105, 163]

$$\mathcal{S}_p\left(\xi, \frac{1}{\beta}\right) = \max(|\xi| - \beta^{p-2}|\xi|^{p-1}, 0)\mathrm{sign}(\xi)$$

to solve the third subproblem

$$\min_h \left\{\alpha_0 \|h\|_p^p + \frac{\eta_0}{2}\|h - \tilde{h}^{(k)} - (\nabla v^{(k+1)} - w^{(k+1)})\|_2^2\right\}$$

and obtain:

$$h^{(k+1)} = \mathcal{S}_p\left((\nabla v^{(k+1)} - w^{(k+1)}) + \tilde{h}^{(k)}, \frac{\alpha_0}{\eta_0}\right)$$

The fourth subproblem

$$\min_s \left\{\alpha_a \|s\|_p^p + \frac{\eta_1}{2}\|s - \tilde{s}^{(k)} - \boldsymbol{\varepsilon}(w^{(k+1)})\|_2^2\right\}$$

can be solved in the same way by

$$s^{(k+1)} = \mathcal{S}_p \left(\boldsymbol{\varepsilon}(w^{(k+1)}) + \tilde{s}^{(k)}, \frac{\alpha_1}{\eta_1} \right).$$

Lastly, the update of the split-Bregman variables is given by

$$\begin{aligned}
\tilde{h}_{x|i,j}^{(k+1)} =& \tilde{h}_{x|i,j}^{(k)} + \left(v_{i+1,j}^{(k+1)} - v_{i,j}^{(k+1)} - w_{x|i,j}^{(k+1)} - h_{x|i,j}^{(k+1)} \right), \\
\tilde{h}_{y|i,j}^{(k+1)} =& \tilde{h}_{y|i,j}^{(k)} + \left(v_{i,j+1}^{(k+1)} - v_{i,j}^{(k+1)} - w_{y|i,j}^{(k+1)} - h_{y|i,j}^{(k+1)} \right),
\end{aligned}$$

and

$$\begin{aligned}
\tilde{s}_{xx|i,j}^{(k+1)} =& \tilde{s}_{xx|i,j}^{(k)} + \left(w_{x|i+1,j}^{(k+1)} - w_{x|i,j}^{(k+1)} - s_{xx|i,j}^{(k+1)} \right), \\
\tilde{s}_{yy|i,j}^{(k+1)} =& \tilde{s}_{yy|i,j}^{(k)} + \left(w_{y|i,j+1}^{(k+1)} - w_{y|i,j}^{(k+1)} - s_{xy|i,j}^{(k+1)} \right), \\
\tilde{s}_{xy|i,j}^{(k+1)} =& \tilde{s}_{xy|i,j}^{(k)} + \left(0.5 \cdot (w_{x|i,j+1}^{(k+1)} - w_{x|i,j}^{(k+1)} + w_{y|i+1,j}^{(k+1)} - w_{y|i,j}^{(k+1)}) - s_{xx|i,j}^{(k+1)} \right).
\end{aligned}$$

TV regularization

Using the TV regularization leads to the following optimization problem:

$$m^* = \min_m \left\{ \frac{1}{2} \| u(m) - u_0 \|_2^2 + \lambda_{\mathrm{TV}} \| m \|_{\mathrm{TV}} \right\}$$

We introduce an auxiliary model v and rewrite the above optimization problem into the following constrained optimization problem

$$\begin{aligned}
m^* =& \min_m \left\{ \frac{1}{2} \| u(m) - u_0 \|_2^2 + \lambda_{\mathrm{TV}} \| v \|_{\mathrm{TV}} \right\} \\
\text{s. t. } m =& v
\end{aligned}$$

Using the alternating direction method of multipliers (ADMM) procedure [30] leads to following three equations:

$$\begin{aligned}
m^{(k+1)} =& \min_m \left\{ \frac{1}{2} \| u(m) - u_0 \|_2^2 + \lambda \| m - v^{(k)} + q^{(k)} \|_2^2 \right\} \\
v^{(k+1)} =& \min_v \left\{ \lambda \| m^{(k+1)} - v + q^{(k)} \|_2^2 + \lambda \| v \|_{\mathrm{TV}} \right\} \\
q^{(k+1)} =& q^{(k+1)} + m^{(k+1)} - v^{(k+1)}
\end{aligned}$$

The superscript (k) refers to the k-th iteration of the FWI algorithm and q is the ADMM variable. We define $g^{(k+1)} = m^{(k+1)} + q^{(k)}$ and introduce two auxiliary variables $w_x \approx \nabla_x u$,

$w_y \approx \nabla_y u$. Substituting these variables in

$$v^{(k+1)} = \min_v \left\{ \lambda \|g^{(k+1)} - v\|_2^2 + \lambda \|v\|_{\mathrm{TV}} \right\}$$

leads to

$$\{v^{(k+1)}, w_x^{(k+1)}, w_y^{(k+1)}\} = \min_{v, w_x, w_y} \left\{ \|g^{(k+1)} - v\|_2^2 + \lambda_0 \|v\|_{\mathrm{TV}} + \alpha \|w_x - \nabla_x v\|_2^2 + \alpha \|w_y - \nabla_y v\|_2^2 \right\}$$

$$.$$

Using the split-Bregman iteration technique [63], we rewrite this into

$$\begin{aligned}\{v^{(k+1)}, w_x^{(k+1)}, w_y^{(k+1)}\} = \min_{v, w_x, w_y} & \left\{ \|g^{(k+1)} - v\|_2^2 + \lambda_0 \|v\|_{\mathrm{TV}} \right. \\ & \left. + \alpha \|w_x - \nabla_x v - b_x^{(k)}\|_2^2 + \alpha \|w_y - \nabla_y v - b_y^{(k)}\|_2^2 \right\}.\end{aligned}$$

where $b_x^{(k+1)} = b_x^{(k)} + (\nabla_x v^{(l+1)} - w_x^{(k+1)})$ and $b_y^{(k+1)} = b_y^{(k)} + (\nabla_y v^{(l+1)} - w_y^{(k+1)})$, with initial values $b_x^{(0)} = b_y^{(0)} = 0$. Employing an alternating minimization algorithm result in the following to subproblems:

$$\begin{aligned} v^{(k+1)} &= \min_v \left\{ \|v - g^{(k+1)}\|_2^2 + \alpha \|w_x^{(k)} - \nabla_x v - b_x^{(k)}\|_2^2 + \alpha \|w_y^{(k)} - \nabla_y v - b_y^{(k)}\|_2^2 \right\} \\ \{w_x, w_y\} &= \min_{w_x, w_y} \left\{ \lambda \|v^{(k+1)}\|_{\mathrm{TV}} + \alpha \|w_x - \nabla_x v^{(k+1)} - b_x^{(k)}\|_2^2 + \alpha \|w_y - \nabla_y v^{(k+1)} - b_y^{(k)}\|_2^2 \right\} \end{aligned}$$

The first-order optimality condition of the first subproblem is

$$(I - \alpha \Delta) v^{(k+1)} = g^{(k+1)} + \alpha \nabla_x^\top (w_x^{(k)} - b_x^{(k)}) + \alpha \nabla_y^\top (w_y^{(k)} - b_y^{(k)}).$$

Using the Gauss-Seidel iteration method yields

$$\begin{aligned} v_{i.j}^{(k+1)} = & \frac{\alpha}{1 + 4\alpha} \left[v_{i+1,j}^{(k)} + v_{i-1,j}^{(k)} + v_{i,j+1}^{(k)} + v_{i,j-1}^{(k)} \right. \\ & + w_{x|i-1,j}^{(k)} - w_{x|i,j}^{(k)} + w_{y|i,j-1}^{(k)} - w_{y|i,j}^{(k)} \\ & \left. - b_{x|i-1,j}^{(k)} + b_{x|i,j}^{(k)} - b_{y|i,j-1}^{(k)} + b_{y|i,j}^{(k)} \right] \\ & + \frac{1}{1 + 4\alpha} g_{i,j}^{(k+1)}, \end{aligned}$$

where (i, j) is the index of a spatial grid point.

The second problem can be solved by using a generalized shrinkage formula [105, 163]:

$$\begin{aligned}
w_x^{(k+1)} &= \max\left(q^{(k+1)} - \frac{\lambda}{2\alpha}, 0\right) \frac{\nabla_x v^{(k+1)} + b_x^{(k+1)}}{q^{(k+1)}} \\
w_y^{(k+1)} &= \max\left(q^{(k+1)} - \frac{\lambda}{2\alpha}, 0\right) \frac{\nabla_y v^{(k+1)} + b_y^{(k+1)}}{q^{(k+1)}} \\
q^{(k+1)} &= \sqrt{|\nabla_x v^{(k+1)} + b_x^{(k+1)}|^2 + |\nabla_y v^{(k+1)} + b_y^{(k+1)}|^2}
\end{aligned}$$

A.1.7 Selection of a regularization weight for different regularization methods

To compare the different regularization methods in section 5.6.3, we benchmarked different regularization weights to find a good regularization weight. The resulting data and model misfit for the different regularization weights and regularization methods are listed in the following tables.

Regularization weights for the inversion using unperturbed objective data

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_0}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-20}$	2.669E-10	9.650E+08	3.389E-10	5.474E+08	3.609E-11	6.334E+07	1.998E-11	1.394E+08
$A \cdot 10^{-21}$	9.051E-12	2.489E+07	9.139E-13	2.027E+06	3.966E-13	1.531E+06	1.074E-13	9.874E+05
$A \cdot 10^{-22}$	1.963E-13	9.352E+05	7.821E-14	9.065E+05	5.549E-14	8.017E+05	1.945E-14	7.032E+05
$A \cdot 10^{-23}$	1.888E-14	6.975E+05	9.892E-15	6.298E+05	7.266E-15	5.916E+05	7.644E-15	5.919E+05
$A \cdot 10^{-24}$	3.352E-15	5.018E+05	9.382E-15	6.144E+05	3.512E-15	4.913E+05	3.846E-15	5.089E+05
$A \cdot 10^{-25}$	4.168E-15	5.020E+05	3.016E-15	4.830E+05	3.372E-15	4.795E+05	3.354E-15	4.688E+05
$A \cdot 10^{-26}$	3.394E-15	4.814E+05	3.356E-15	4.852E+05	3.951E-15	5.007E+05	5.467E-15	4.988E+05
$A \cdot 10^{-27}$	3.438E-15	4.983E+05	3.214E-15	4.822E+05	4.388E-15	4.993E+05	3.662E-15	4.699E+05
$A \cdot 10^{-28}$	3.510E-15	4.666E+05	3.028E-15	4.810E+05	3.910E-15	4.818E+05	3.998E-15	4.925E+05
$A \cdot 10^{-29}$	3.172E-15	4.849E+05	3.297E-15	4.893E+05	3.439E-15	4.908E+05	3.772E-15	4.707E+05
$A \cdot 10^{-30}$	2.793E-15	4.714E+05	2.916E-15	4.734E+05	3.032E-15	4.801E+05	2.960E-15	4.778E+05
Iteration 100								
$A \cdot 10^{-20}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
$A \cdot 10^{-21}$	N/A	N/A	N/A	N/A	N/A	N/A	1.130E-13	9.886E+05
$A \cdot 10^{-22}$	N/A	N/A	1.324E-13	9.745E+05	1.472E-13	9.851E+05	2.470E-13	9.156E+05
$A \cdot 10^{-23}$	2.060E-13	8.943E+05	1.506E-13	9.028E+05	2.046E-13	8.774E+05	2.468E-13	1.066E+06
$A \cdot 10^{-24}$	2.050E-13	1.066E+06	1.740E-13	1.025E+06	2.116E-13	1.091E+06	1.714E-13	8.644E+05
$A \cdot 10^{-25}$	1.753E-13	8.772E+05	2.910E-13	1.011E+06	1.866E-13	1.059E+06	3.111E-13	1.106E+06
$A \cdot 10^{-26}$	2.430E-13	1.076E+06	2.916E-13	1.117E+06	2.456E-13	1.098E+06	1.712E-13	8.696E+05
$A \cdot 10^{-27}$	2.864E-13	1.022E+06	2.389E-13	1.061E+06	3.094E-13	1.204E+06	2.804E-10	1.039E+06
$A \cdot 10^{-28}$	1.550E-13	8.987E+05	2.318E-13	1.025E+06	2.647E-13	1.118E+06	3.173E-13	1.186E+06
$A \cdot 10^{-29}$	2.818E-13	1.086E+06	2.089E-13	9.289E+05	2.806E-13	1.120E+06	3.131E-13	1.200E+06
$A \cdot 10^{-30}$	2.203E-13	9.846E+05	2.644E-13	1.127E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06

Table A.1: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_0}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using unperturbed objective data. N/A' describes that the algorithm was terminated before the 100 iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_1}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-20}$	1.926E-13	1.005E+06	1.841E-13	8.959E+05	2.975E-14	7.273E+05	2.955E-14	7.231E+05
$A \cdot 10^{-21}$	1.708E-14	6.756E+05	1.790E-14	6.712E+05	4.693E-15	5.355E+05	4.873E-15	5.303E+05
$A \cdot 10^{-22}$	3.949E-15	5.051E+05	3.315E-15	4.927E+05	4.544E-15	4.904E+05	2.653E-15	4.785E+05
$A \cdot 10^{-23}$	3.229E-15	4.831E+05	3.613E-15	4.903E+05	2.949E-15	4.819E+05	3.147E-15	4.804E+05
$A \cdot 10^{-24}$	3.493E-15	4.769E+05	3.087E-15	4.741E+05	3.117E-15	4.763E+05	3.306E-15	4.850E+05
$A \cdot 10^{-25}$	2.955E-15	4.713E+05	2.517E-15	4.559E+05	2.813E-15	4.735E+05	3.247E-15	4.704E+05
$A \cdot 10^{-26}$	3.893E-15	5.094E+05	3.623E-15	4.730E+05	2.710E-15	4.680E+05	4.215E-15	4.885E+05
$A \cdot 10^{-27}$	2.663E-15	4.682E+05	3.689E-15	4.829E+05	3.744E-15	4.885E+05	4.175E-15	5.000E+05
$A \cdot 10^{-28}$	3.803E-15	5.131E+05	3.309E-15	4.750E+05	2.866E-15	4.622E+05	3.439E-15	4.713E+05
$A \cdot 10^{-29}$	3.325E-15	4.946E+05	3.025E-15	4.737E+05	3.670E-15	4.857E+05	2.735E-15	4.597E+05
$A \cdot 10^{-30}$	2.668E-15	4.551E+05	2.668E-15	4.551E+05	2.668E-15	4.551E+05	2.668E-15	4.551E+05
Iteration 100								
$A \cdot 10^{-20}$	N/A	N/A	N/A	N/A	2.157E-13	9.521E+05	2.161E-13	1.084E+06
$A \cdot 10^{-21}$	2.376E-13	1.129E+06	1.837E-13	9.972E+05	3.541E-13	1.253E+06	3.379E-13	1.223E+06
$A \cdot 10^{-22}$	2.536E-13	1.064E+06	2.225E-13	1.071E+06	2.165E-13	1.076E+06	2.402E-13	1.120E+06
$A \cdot 10^{-23}$	3.322E-13	1.044E+06	2.732E-13	9.627E+05	1.999E-13	9.342E+05	3.182E-13	1.014E+06
$A \cdot 10^{-24}$	2.090E-13	9.414E+05	2.724E-13	1.086E+06	2.330E-13	1.089E+06	2.682E-13	1.068E+06
$A \cdot 10^{-25}$	3.542E-13	1.231E+06	2.125E-13	1.059E+06	2.414E-13	8.916E+05	3.409E-13	1.168E+09
$A \cdot 10^{-26}$	2.337E-13	1.075E+06	2.865E-13	1.177E+06	2.903E-13	1.177E+06	2.064E-13	1.013E+06
$A \cdot 10^{-27}$	2.899E-13	1.142E+06	2.925E-13	1.106E+06	2.240E-13	1.077E+06	2.812E-13	1.111E+06
$A \cdot 10^{-28}$	2.320E-13	1.081E+06	2.780E-13	1.064E+06	2.645E-13	1.128E+06	2.669E-13	9.805E+05
$A \cdot 10^{-29}$	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06
$A \cdot 10^{-30}$	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06

Table A.2: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_1}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using unperturbed objective data. N/A' describes that the algorithm was terminated before the 100 iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_2}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-20}$	2.888E-13	1.110E+06	1.666E-13	1.044E+06	2.034E-13	9.328E+05	4.475E-14	8.023E+05
$A \cdot 10^{-21}$	2.315E-14	7.156E+05	2.560E-14	7.298E+05	1.264E-14	6.382E+05	7.412E-15	5.794E+05
$A \cdot 10^{-22}$	7.894E-15	5.728E+05	5.721E-15	5.462E+05	3.226E-15	4.954E+05	3.964E-15	5.151E+05
$A \cdot 10^{-23}$	3.193E-15	4.895E+05	2.993E-15	4.860E+05	3.325E-15	4.696E+05	3.920E-15	4.990E+05
$A \cdot 10^{-24}$	3.397E-15	4.989E+05	3.487E-15	4.927E+05	3.776E-15	4.913E+05	2.748E-15	4.693E+05
$A \cdot 10^{-25}$	4.029E-15	4.916E+05	2.770E-15	4.730E+05	2.818E-15	4.771E+05	2.947E-15	4.791E+05
$A \cdot 10^{-26}$	3.692E-15	5.130E+05	3.725E-15	4.920E+05	3.060E-15	4.702E+05	4.365E-15	4.981E+05
$A \cdot 10^{-27}$	3.449E-15	4.632E+05	3.389E-15	4.791E+05	2.985E-15	4.738E+05	4.813E-15	5.425E+05
$A \cdot 10^{-28}$	3.974E-15	4.943E+05	3.348E-15	4.835E+05	3.600E-15	4.727E+05	3.555E-15	4.851E+05
$A \cdot 10^{-29}$	4.017E-15	4.994E+05	4.615E-15	5.055E+05	3.031E-15	4.671E+05	2.899E-15	4.755E+05
$A \cdot 10^{-30}$	3.343E-15	4.686E+05	2.865E-15	4.721E+05	2.668E-15	4.551E+05	2.668E-15	4.551E+05
Iteration 100								
$A \cdot 10^{-20}$	N/A	N/A	N/A	N/A	N/A	N/A	3.232E-13	1.207E+06
$A \cdot 10^{-21}$	1.654E-13	9.498E+05	3.409E-13	1.241E+06	1.585E-13	9.843E+05	2.582E-13	1.117E+06
$A \cdot 10^{-22}$	2.215E-13	1.112E+06	2.436E-13	1.052E+06	2.785E-13	1.136E+06	2.908E-13	1.170E+06
$A \cdot 10^{-23}$	2.423E-13	1.137E+06	1.305E-13	9.487E+05	2.706E-13	1.088E+06	3.394E-13	1.235E+06
$A \cdot 10^{-24}$	1.735E-13	9.460E+05	2.512E-13	1.037E+06	1.765E-13	9.863E+05	2.175E-13	9.184E+05
$A \cdot 10^{-25}$	2.309E-13	9.646E+05	2.699E-13	1.103E+06	2.386E-13	1.077E+06	2.778E-13	1.164E+06
$A \cdot 10^{-26}$	1.886E-13	9.814E+05	2.182E-13	1.053E+06	3.038E-13	1.021E+06	2.219E-13	1.048E+06
$A \cdot 10^{-27}$	2.989E-13	1.186E+06	3.081E-13	1.183E+06	2.869E-13	1.189E+06	3.044E-13	1.204E+06
$A \cdot 10^{-28}$	1.725E-13	9.046E+05	3.070E-13	1.135E+06	2.208E-13	1.109E+06	3.347E-13	1.214E+06
$A \cdot 10^{-29}$	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06
$A \cdot 10^{-30}$	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06	3.347E-13	1.214E+06

Table A.3: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_2}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using unperturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_{\mathcal{T}_p}}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-1}$	3.850E-11	6.636E+06	6.259E-12	2.839E+06	1.489E-11	4.563E+06	3.372E-13	1.167E+06
$A \cdot 10^{-2}$	4.070E-13	1.232E+06	3.389E-13	1.172E+06	4.640E-13	1.266E+06	2.931E-13	1.084E+06
$A \cdot 10^{-3}$	1.586E-13	9.851E+05	1.108E-13	8.965E+05	2.315E-14	7.077E+05	2.025E-14	6.994E+05
$A \cdot 10^{-4}$	2.022E-14	6.378E+05	7.008E-15	5.735E+05	7.768E-15	5.903E+05	5.938E-15	5.100E+05
$A \cdot 10^{-5}$	5.176E-15	5.148E+05	3.919E-15	5.191E+05	4.472E-15	4.928E+05	2.953E-15	4.842E+05
$A \cdot 10^{-6}$	2.689E-15	4.620E+05	3.081E-15	4.758E+05	3.645E-15	4.785E+05	3.855E-15	5.064E+05
$A \cdot 10^{-7}$	2.923E-15	4.574E+05	2.848E-15	4.586E+05	3.799E-15	4.866E+05	3.343E-15	4.888E+05
$A \cdot 10^{-8}$	3.223E-15	4.662E+05	3.379E-15	4.978E+05	3.741E-15	4.860E+05	3.313E-15	4.797E+05
$A \cdot 10^{-9}$	3.336E-15	4.775E+05	3.790E-15	4.991E+05	2.800E-15	4.607E+05	3.918E-15	4.976E+05
$A \cdot 10^{-10}$	2.913E-15	4.717E+05	4.860E-15	5.028E+05	4.038E-15	4.933E+05	3.141E-15	4.638E+05
Iteration 100								
$A \cdot 10^{-1}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-2}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-3}$	1.600E-13	9.870E+05	1.563E-13	9.907E+05	1.329E-13	9.646E+05	1.510E-13	9.455E+05
$A \cdot 10^{-4}$	1.913E-13	8.677E+05	1.915E-13	9.183E+05	1.530E-13	9.196E+05	2.017E-13	8.658E+05
$A \cdot 10^{-5}$	1.511E-13	9.847E+05	2.303E-13	1.062E+06	2.306E-13	1.055E+06	3.106E-13	1.073E+06
$A \cdot 10^{-6}$	2.483E-13	1.064E+06	2.293E-13	1.042E+06	2.634E-13	1.071E+06	2.480E-13	9.432E+05
$A \cdot 10^{-7}$	2.486E-13	1.070E+06	2.324E-13	9.773E+05	2.426E-13	1.060E+06	1.469E-13	9.652E+05
$A \cdot 10^{-8}$	3.237E-13	1.180E+06	3.291E-13	1.193E+06	1.996E-13	9.257E+05	2.627E-13	1.132E+06
$A \cdot 10^{-9}$	2.776E-13	1.008E+06	2.206E-13	1.030E+06	1.772E-13	1.024E+06	2.474E-13	1.101E+06
$A \cdot 10^{-10}$	2.159E-13	1.039E+06	1.971E-13	8.885E+05	2.345E-13	1.081E+06	3.208E-13	1.088E+06

Table A.4: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_{\mathcal{T}_p}}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using unperturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_{\text{TV}}}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-1}$	3.727E-11	6.591E+06	1.928E-11	5.223E+06	3.058E-12	2.158E+06	4.571E-13	1.248E+06
$A \cdot 10^{-2}$	3.349E-13	1.179E+06	3.555E-13	1.201E+06	3.138E-13	1.104E+06	2.090E-13	9.767E+05
$A \cdot 10^{-3}$	1.478E-13	8.837E+05	5.776E-14	8.013E+05	4.015E-14	7.783E+05	2.253E-14	6.529E+05
$A \cdot 10^{-4}$	1.553E-14	5.880E+05	5.515E-15	5.208E+05	5.925E-15	5.279E+05	3.785E-15	4.840E+05
$A \cdot 10^{-5}$	4.485E-15	4.857E+05	3.192E-15	4.699E+05	4.335E-15	4.897E+05	4.790E-15	5.063E+05
$A \cdot 10^{-6}$	3.208E-15	4.852E+05	3.187E-15	4.709E+05	2.830E-15	4.733E+05	2.720E-15	4.617E+05
$A \cdot 10^{-7}$	4.669E-15	5.017E+05	3.343E-15	4.970E+05	3.072E-15	4.736E+05	2.768E-15	4.560E+05
$A \cdot 10^{-8}$	4.029E-15	5.102E+05	4.829E-15	5.046E+05	3.914E-15	4.783E+05	3.358E-15	4.935E+05
$A \cdot 10^{-9}$	3.133E-15	4.696E+05	2.872E-15	4.659E+05	3.116E-15	4.699E+05	4.024E-15	4.966E+05
$A \cdot 10^{-10}$	3.653E-15	4.878E+05	3.823E-15	4.837E+05	5.384E-15	5.185E+05	2.789E-15	4.663E+05
Iteration 100								
$A \cdot 10^{-1}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-2}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-3}$	1.546E-13	8.933E+05	1.552E-13	8.933E+05	1.159E-13	9.012E+05	1.712E-13	9.232E+05
$A \cdot 10^{-4}$	1.627E-13	9.764E+05	1.966E-13	1.012E+06	1.532E-13	1.021E+06	1.334E-13	9.640E+05
$A \cdot 10^{-5}$	1.535E-13	9.209E+05	2.080E-13	1.026E+06	2.722E-13	1.102E+06	1.834E-13	9.209E+05
$A \cdot 10^{-6}$	2.273E-13	1.050E+06	2.253E-13	9.189E+05	1.977E-13	9.545E+05	2.775E-13	1.163E+06
$A \cdot 10^{-7}$	2.275E-13	9.794E+05	1.464E-13	8.941E+05	3.252E-13	1.221E+06	2.123E-13	9.441E+05
$A \cdot 10^{-8}$	1.708E-13	8.454E+05	2.657E-13	1.094E+06	1.918E-13	1.048E+06	3.439E-13	1.230E+06
$A \cdot 10^{-9}$	1.468E-13	9.435E+05	2.322E-13	1.112E+06	2.796E-13	1.001E+06	2.772E-13	1.064E+06
$A \cdot 10^{-10}$	2.978E-13	1.166E+06	1.831E-13	9.425E+05	2.979E-13	1.190E+06	2.667E-13	9.745E+05

Table A.5: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_{\text{TV}}}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using unperturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

Regularization weights for the inversion using perturbed objective data

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_0}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-20}$	3.974E-10	2.864E+09	1.230E-10	1.124E+07	2.108E-10	1.090E+09	9.765E-11	2.142E+07
$A \cdot 10^{-21}$	9.530E-11	5.889E+06	9.725E-11	8.222E+06	9.445E-11	1.333E+06	9.418E-11	1.380E+06
$A \cdot 10^{-22}$	9.422E-11	1.877E+06	9.414E-11	1.143E+06	9.422E-11	1.163E+09	9.415E-11	1.145E+06
$A \cdot 10^{-23}$	9.413E-11	1.532E+06	9.407E-11	1.245E+06	9.411E-11	1.118E+06	9.410E-11	1.203E+06
$A \cdot 10^{-24}$	9.415E-11	1.112E+06	9.409E-11	1.148E+06	9.416E-11	1.124E+06	9.406E-08	1.278E+06
$A \cdot 10^{-25}$	9.417E-11	1.138E+06	9.417E-11	1.147E+06	9.417E-11	1.116E+06	9.425E-11	1.130E+06
$A \cdot 10^{-26}$	9.416E-11	1.154E+06	9.416E-11	1.120E+06	9.418E-11	1.161E+06	9.409E-11	1.129E+06
$A \cdot 10^{-27}$	9.414E-11	1.123E+06	9.406E-11	1.255E+06	9.415E-11	1.150E+06	9.412E-11	1.183E+06
$A \cdot 10^{-28}$	9.414E-11	1.143E+06	9.420E-11	1.109E+06	9.408E-11	1.141E+06	9.418E-11	1.114E+06
$A \cdot 10^{-29}$	9.410E-11	1.108E+06	9.416E-11	1.151E+06	9.419E-11	1.145E+06	9.410E-11	1.135E+06
$A \cdot 10^{-30}$	9.406E-11	1.335E+06	9.410E-11	1.125E+06	9.408E-11	1.125E+06	9.427E-11	1.116E+06
Iteration 100								
$A \cdot 10^{-20}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
$A \cdot 10^{-21}$	N/A	N/A	N/A	N/A	N/A	N/A	9.424E-11	1.172E+06
$A \cdot 10^{-22}$	9.416E-11	1.168E+06	9.423E-11	1.136E+06	9.421E-11	1.175E+06	9.420E-11	1.167E+06
$A \cdot 10^{-23}$	9.435E-11	1.137E+06	9.417E-11	1.161E+06	9.416E-11	1.122E+06	9.413E-11	1.104E+06
$A \cdot 10^{-24}$	N/A	N/A	9.415E-11	1.117E+06	N/A	N/A	9.415E-11	1.132E+06
$A \cdot 10^{-25}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
$A \cdot 10^{-26}$	9.416E-11	1.154E+06	9.416E-11	1.110E+06	9.418E-11	1.161E+06	9.418E-11	1.158E+06
$A \cdot 10^{-27}$	9.415E-11	1.129E+06	9.417E-11	1.124E+06	9.416E-11	1.135E+06	9.415E-11	1.130E+06
$A \cdot 10^{-28}$	9.418E-11	1.131E+06	N/A	N/A	9.411E-11	1.112E+06	N/A	
$A \cdot 10^{-29}$	9.418E-11	1.122E+06	9.416E-11	1.151E+06	N/A	N/A	9.416E-11	1.133E+06
$A \cdot 10^{-30}$	9.417E-11	1.153E+06	9.414E-11	1.120E+06	9.416E-11	1.111E+06	N/A	

Table A.6: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_0}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using perturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_1}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-20}$	9.436E-11	1.237E+06	9.424E-11	1.207E+06	9.423E-11	1.191E+06	9.425E-11	1.118E+06
$A \cdot 10^{-21}$	9.427E-11	1.111E+06	9.417E-11	1.116E+06	9.414E-11	1.128E+06	9.406E-11	1.205E+06
$A \cdot 10^{-22}$	9.414E-11	1.121E+06	9.410E-11	1.195E+06	9.422E-11	1.165E+06	9.412E-11	1.114E+06
$A \cdot 10^{-23}$	9.409E-11	1.175E+06	9.409E-11	1.143E+06	9.417E-11	1.127E+06	9.420E-11	1.178E+06
$A \cdot 10^{-24}$	9.409E-11	1.188E+06	9.409E-11	1.143E+06	9.425E-11	1.131E+06	9.417E-11	1.148E+06
$A \cdot 10^{-25}$	9.412E-11	1.124E+06	9.418E-11	1.116E+06	9.415E-11	1.197E+06	9.419E-11	1.140E+06
$A \cdot 10^{-26}$	9.408E-11	1.136E+06	9.416E-11	1.125E+06	9.413E-11	1.107E+06	9.409E-11	1.146E+06
$A \cdot 10^{-27}$	9.408E-11	1.137E+06	9.409E-11	1.144E+06	9.416E-11	1.160E+06	9.417E-11	1.134E+06
$A \cdot 10^{-28}$	9.425E-11	1.137E+09	9.414E-11	1.130E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06
$A \cdot 10^{-29}$	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06
$A \cdot 10^{-30}$	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06
Iteration 100								
$A \cdot 10^{-20}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
$A \cdot 10^{-21}$	N/A	N/A	N/A	N/A	9.416E-11	1.130E+06	9.419E-11	1.108E+06
$A \cdot 10^{-22}$	9.419E-11	1.154E+06	9.412E-11	1.126E+06	9.422E-11	1.139E+06	9.416E-11	1.135E+06
$A \cdot 10^{-23}$	9.414E-11	1.111E+06	9.416E-11	1.149E+06	9.421E-11	1.191E+06	9.422E-11	1.197E+06
$A \cdot 10^{-24}$	9.414E-11	1.126E+06	9.414E-11	1.117E+06	N/A	N/A	9.417E-11	1.143E+06
$A \cdot 10^{-25}$	9.421E-11	1.115E+06	N/A	N/A	9.416E-11	1.131E+06	N/A	
$A \cdot 10^{-26}$	9.415E-11	1.119E+06	9.421E-11	1.172E+06	9.419E-11	1.107E+06	9.416E-11	1.103E+06
$A \cdot 10^{-27}$	9.424E-11	1.173E+06	9.418E-11	1.119E+06	9.422E-11	1.112E+06	N/A	
$A \cdot 10^{-28}$	N/A	N/A	9.416E-11	1.121E+06	N/A	N/A	N/A	
$A \cdot 10^{-29}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
$A \cdot 10^{-30}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Table A.7: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_1}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using perturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_2}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-20}$	9.439E-11	1.265E+06	9.448E-11	1.345E+06	9.453E-11	1.380E+06	9.432E-11	1.185E+06
$A \cdot 10^{-21}$	9.412E-11	1.127E+06	9.416E-11	1.110E+06	9.427E-11	1.137E+06	9.412E-11	1.128E+06
$A \cdot 10^{-22}$	9.414E-11	1.117E+06	9.420E-11	1.110E+06	9.408E-11	1.172E+06	9.417E-11	1.144E+06
$A \cdot 10^{-23}$	9.418E-11	1.118E+06	9.415E-11	1.170E+06	9.426E-11	1.171E+06	9.406E-11	1.170E+06
$A \cdot 10^{-24}$	9.408E-11	1.268E+06	9.415E-11	1.120E+06	9.410E-11	1.114E+06	9.416E-11	1.109E+06
$A \cdot 10^{-25}$	9.408E-11	1.116E+06	9.408E-11	1.152E+06	9.413E-11	1.125E+06	9.407E-11	1.177E+06
$A \cdot 10^{-26}$	9.409E-11	1.243E+06	9.411E-11	1.194E+06	9.417E-11	1.157E+06	9.427E-11	1.110E+06
$A \cdot 10^{-27}$	9.412E-11	1.165E+06	9.414E-11	1.136E+06	9.408E-11	1.129E+06	9.421E-11	1.155E+06
$A \cdot 10^{-28}$	9.411E-11	1.197E+06	9.417E-11	1.200E+06	9.418E-11	1.125E+06	9.420E-11	1.142E+06
$A \cdot 10^{-29}$	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06
$A \cdot 10^{-30}$	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06	9.420E-11	1.142E+06
Iteration 100								
$A \cdot 10^{-20}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
$A \cdot 10^{-21}$	N/A	N/A	N/A	N/A	9.415E-11	1.112E+06	9.416E-11	1.113E+06
$A \cdot 10^{-22}$	9.421E-11	1.106E+06	N/A	N/A	9.415E-11	1.117E+06	9.417E-11	1.144E+06
$A \cdot 10^{-23}$	N/A	N/A	9.416E-11	1.120E+06	9.418E-11	1.126E+06	9.425E-11	1.156E+06
$A \cdot 10^{-24}$	9.415E-11	1.113E+06	9.417E-11	1.152E+06	9.417E-11	1.112E+06	9.420E-11	1.146E+06
$A \cdot 10^{-25}$	9.413E-11	1.122E+06	9.414E-11	1.143E+06	9.416E-11	1.119E+06	9.414E-11	1.103E+06
$A \cdot 10^{-26}$	9.417E-11	1.157E+06	9.415E-11	1.104E+06	9.418E-11	1.139E+06	N/A	
$A \cdot 10^{-27}$	9.413E-11	1.128E+06	9.416E-11	1.122E+06	9.417E-11	1.125E+06	N/A	
$A \cdot 10^{-28}$	N/A	N/A	9.417E-11	1.176E+06	9.416E-11	1.122E+06	N/A	
$A \cdot 10^{-29}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
$A \cdot 10^{-30}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Table A.8: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_2}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using perturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_{\mathcal{T}_p}}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-1}$	1.279E-10	6.406E+06	1.168E-10	5.616E+06	9.593E-11	1.908E+06	9.437E-11	1.237E+06
$A \cdot 10^{-2}$	9.446E-11	1.289E+06	9.444E-11	1.294E+06	9.447E-11	1.257E+06	9.440E-11	1.153E+06
$A \cdot 10^{-3}$	9.428E-11	1.117E+06	9.427E-11	1.127E+06	9.417E-11	1.122E+06	9.418E-11	1.161E+06
$A \cdot 10^{-4}$	9.421E-11	1.118E+06	9.420E-11	1.131E+06	9.420E-11	1.113E+06	9.422E-11	1.106E+06
$A \cdot 10^{-5}$	9.406E-11	1.212E+06	9.412E-11	1.125E+06	9.415E-11	1.119E+06	9.419E-11	1.141E+06
$A \cdot 10^{-6}$	9.420E-11	1.122E+06	9.408E-11	1.152E+06	9.412E-11	1.106E+06	9.412E-11	1.130E+06
$A \cdot 10^{-7}$	9.417E-11	1.198E+06	9.416E-11	1.134E+06	9.418E-11	1.149E+06	9.414E-11	1.138E+09
$A \cdot 10^{-8}$	9.418E-11	1.139E+06	9.422E-11	1.113E+06	9.414E-11	1.108E+06	9.408E-11	1.120E+06
$A \cdot 10^{-9}$	9.419E-11	1.121E+06	9.413E-11	1.108E+06	9.419E-11	1.151E+06	9.412E-11	1.198E+06
$A \cdot 10^{-10}$	9.409E-11	1.115E+06	9.419E-11	1.154E+06	9.411E-11	1.112E+06	9.406E-11	1.180E+06
Iteration 100								
$A \cdot 10^{-1}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-2}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-3}$	9.428E-11	1.117E+06	N/A	N/A	N/A	N/A	9.420E-11	1.174E+06
$A \cdot 10^{-4}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-5}$	9.415E-11	1.125E+06	9.413E-11	1.109E+06	9.415E-11	1.109E+06	9.420E-11	1.137E+06
$A \cdot 10^{-6}$	N/A	N/A	9.415E-11	1.103E+06	9.420E-11	1.120E+06	9.414E-11	1.123E+06
$A \cdot 10^{-7}$	9.418E-11	1.153E+06	9.416E-11	1.124E+06	N/A	N/A	9.417E-11	1.133E+06
$A \cdot 10^{-8}$	N/A	N/A	9.423E-11	1.131E+06	9.415E-11	1.110E+06	9.412E-11	1.112E+06
$A \cdot 10^{-9}$	N/A	N/A	9.418E-11	1.124E+06	N/A	N/A	9.414E-11	1.115E+06
$A \cdot 10^{-10}$	9.416E-11	1.135E+06	9.420E-11	1.151E+06	9.417E-11	1.105E+06	9.413E-11	1.103E+06

Table A.9: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_{\mathcal{T}_p}}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using perturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

Last iteration								
$A =$	7.5		5		2.5		1	
$\lambda_{\mathcal{R}_{\mathrm{TV}}}$	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit	data misfit	model misfit
$A \cdot 10^{-1}$	1.255E-10	6.306E+06	9.948E-11	2.788E+06	9.624E-11	1.958E+06	9.434E-11	1.210E+06
$A \cdot 10^{-2}$	1.255E-10	6.306E+06	9.438E-11	1.253E+06	9.440E-11	1.198E+06	9.427E-11	1.127E+06
$A \cdot 10^{-3}$	9.431E-11	1.183E+06	9.415E-11	1.117E+06	9.423E-11	1.115E+06	9.412E-11	1.148E+06
$A \cdot 10^{-4}$	9.442E-11	1.183E+06	9.410E-11	1.130E+06	9.420E-11	1.149E+06	9.421E-11	1.164E+06
$A \cdot 10^{-5}$	9.411E-11	1.219E+06	9.418E-11	1.131E+06	9.409E-11	1.195E+06	9.411E-11	1.142E+06
$A \cdot 10^{-6}$	9.413E-11	1.161E+06	9.418E-11	1.165E+06	9.415E-11	1.307E+06	9.412E-11	1.126E+06
$A \cdot 10^{-7}$	9.417E-11	1.131E+06	9.414E-11	1.129E+06	9.413E-11	1.151E+06	9.421E-11	1.118E+06
$A \cdot 10^{-8}$	9.418E-11	1.137E+06	9.425E-11	1.129E+06	9.426E-11	1.139E+06	9.422E-11	1.116E+06
$A \cdot 10^{-9}$	9.420E-11	1.448E+06	9.423E-11	1.147E+06	9.409E-11	1.154E+06	9.413E-11	1.147E+06
$A \cdot 10^{-10}$	9.411E-11	1.121E+06	9.406E-11	1.292E+06	9.407E-11	1.162E+06	9.415E-11	1.186E+06
Iteration 100								
$A \cdot 10^{-1}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-2}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$A \cdot 10^{-3}$	N/A	N/A	N/A	N/A	N/A	N/A	9.414E-11	1.113E+06
$A \cdot 10^{-4}$	9.413E-11	1.127E+06	9.411E-11	1.111E+06	N/A	N/A	N/A	N/A
$A \cdot 10^{-5}$	9.415E-11	1.133E+06	N/A	N/A	9.412E-11	1.112E+06	9.414E-11	1.110E+06
$A \cdot 10^{-6}$	N/A	N/A	9.418E-11	1.165E+06	9.419E-11	1.161E+06	9.417E-11	1.152E+06
$A \cdot 10^{-7}$	N/A	N/A	9.415E-11	1.120E+06	9.415E-11	1.119E+06	N/A	N/A
$A \cdot 10^{-8}$	9.424E-11	1.170E+06	N/A	N/A	N/A	N/A	9.422E-11	1.116E+06
$A \cdot 10^{-9}$	9.416E-11	1.141E+06	N/A	N/A	9.416E-11	1.113E+06	9.414E-11	1.125E+06
$A \cdot 10^{-10}$	9.415E-11	1.125E+06	9.418E-11	1.138E+06	9.425E-11	1.139E+06	9.416E-11	1.131E+06

Table A.10: Model and data misfit for different regularization weights $\lambda_{\mathcal{R}_{\mathrm{TV}}}$ after completion of the inverting algorithm (upper half) and after the 100th iteration (lower half), using perturbed objective data. N/A' describes that the algorithm was terminated before the 100th iteration.

A.1.8 Marmousi2 viscoelastic model

In this section, the Marmousi model used in section 5.6.6 is described in more detail. The basis of the model is the model originally designed for acoustic wave propagation by Brougois et al. [37, 161] and the variant for elastic material of Martin [102], which is available online [101]. We consider here only the subdomain of the original Marmousi model and not the extended domain of the Marmousi2 model.

For the transition from elastic to viscoelastic material we follow the procedure of Hamilton [68] and get the Q factor by

$$\frac{1}{Q} = \frac{a \cdot V}{\pi \cdot f - \frac{a^2 \cdot V^2}{4\pi f}}$$

where a is an attenuation coefficient, f is the frequency, and V is the wave velocity. Since we assume constant Q factors, we set the frequency constant to $f = 80$. To obtain lower attenuation in the deeper regions, we choose the attenuation coefficient as $a = \frac{1}{c_1 \cdot V + c_2}$. We set the constants $c_1 = 4.1642$ and $c_2 = -6245.4789$ to obtain values between 605 and 19 for Q_1. To calculate Q_2 we set $c_1 = 2.0419$ and $c_2 = -619.4326$ to obtain values between 390 and 14. This choice gives us a comparable model to the one in [90] and is based on the following two hypothetical assumptions [68, 90, 143]:

- Attenuation increases with decreasing velocity.
- The velocity increases gradually with depth.

Whereby the second assumption is already included in the Marmousi model. Since the resolution of the Marmousi model is very high and need not be so high for the considered frequencies, we reduce the resolution of the model from $1.25m$ between two grid points to $15m$ distance between grid points.

Analogous to the work by Martin [102], we add a vacuum layer of 20 grid points above the model. We need this layer to place the transmitters and receivers. This layer can have different physical properties like those of air, water or in our case vacuum. Furthermore, another 30 grid points are added above this layer, representing the CPML layer. The values of these two layers can be taken from the table A.11. Similarly, CPML layers of 20 grid points each are added to the right, left and bottom, taking the values of the respective edge layer.

	ρ in kg/m^3	v_p in m/s	v_s in m/s	Q_1	Q_2
vacuum	1000	1500	700	60000	60000
CPML	20	300	10	1000	1000

Table A.11: Physical properties of the vacuum and CPML layer.

In total, this results in an area of size $9840m \times 4112m$, corresponding to 653×270 grid points, with a distance of $15m$ between the grid points. We set the origin to the upper left corner of the original Marmousi model.

The sources are set $150m$ above the original Marmousi model (where we add the vacuum layer). The first source is positioned at $(250m, -150m)$. The remaining 31 sources are positioned at a distance of $287m$ along the $y = -150m$ line. The receivers are positioned slightly lower, at $y = -100m$. The first receiver at position $(100m, -100m)$, and the remaining 369 receivers, with a distance of $25m$ along the $y = -100m$ line. The setup is shown in figure A.1.28.

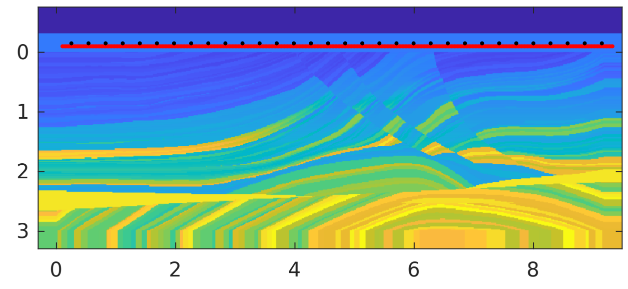


Figure A.1.28: Positions of the sources and receivers for the Marmousi example. The black points describe the positions of the sources, the red points describe the locations of the receivers.

The initial model is created by the convolution of the original Marmousi model and a Gaussian kernel. Then the resolution of the model is reduced and the CMPL and vacuum layers are created as described above. We use Matlab [104] to create the initial model and use the procedure described in listing A.1.

```
sz = 300;
lx = 100;
ly = 200;
H=zeros(sz,sz);
i0=sz/2;
j0=sz/2;
for j=1:1:sz
    for i=1:1:sz
        H(i,j)=exp(-0.5 *( ((i-i0)/lx)^2 + ((j-j0)/ly))^2);
    end
end
H=H/sum(H(:));
init_model = imfilter(true_model, H,'replicate');
```

Listing A.1: Generation of the initial model.