SIMULATION OF NONLINEAR WAVE PROPAGATION IN ULTRASOUND

HÅKON SELJÅSEN

Master Thesis for the degree of
Sivilingeniør / Master of Science

Department of Physics
The Faculty of Natural Sciences and Technology
The Norwegian University of Science and Technology

June 2014 – version 1.0

Supervisors:
Professor Hans Torp[1]
Professor Catharina de Lange Davies[2]

1 The Department of Circulation and Medical Imaging
2 The Department of Physics

Håkon Seljåsen:
Simulation of nonlinear wave propagation in ultrasound.

SUPERVISORS:
Professor Hans Torp
Professor Catharina de Lange Davies

Trondheim,
June 2014

PREFACE

This master thesis is submitted in fulfillment of the requirements for the degree of Sivilingneniør/Master of Science, at The Norwegian University of Science and Technology, Department of Physics.

The given problem is shown in the following quote:

> Discuss the theoretical fundament in different approaches to simulations of nonlinear wave propagation in ultrasound, and make a fast implementation using the quasi-linear approximation, based on the "Propose" method.
>
> Compare the results to a state-of-the-art simulation method; e.g. "Abersim", for a typical cardiac ultrasound probe. The comparison should include both magnitude and phase mapping at multiple depth ranges.

ACKNOWLEDGMENTS

I want to thank:

- Professor Hans Torp at the Department of Circulation and Medical Imaging, for helpful supervising and encouragements during my project work.
- Bjørn Kolbrek at the Department of Acoustics for valuable debugging help and advises on Matlab.
- Randi Seljåsen for structural and linguistic advices on the text.
- Alfonso Rodriguez at the Department of Circulation and Medical Imaging. He has tested the *sinc(x)* implementation used in my specialization project [1], finding results contradictory to the original conclusion. His findings inspired the study that lead to the report in App. A, and he also gave helpful discussions on the matter.

COLOPHON

Any colored text or numbers in this thesis indicate a clickable hyperlink, the colored numbers in the table of contents indicate page numbers. References are given as green numbers inside bold brackets, other hyperlinks are specified as chapter, section, appendix, figure, table or listing. Equations are by default only denoted by their numbers inside soft brackets, but the bracket will be preceded by the word *Equation* in the start of a sentence.

Simulation software packages like *Abersim* and *Propose* are denoted with capital letter, as names and emphasized in *italics*. Sub-routines and other special names which originally does not have capital letter, like *sinc(x)* and *prop2harm*, are emphasized in *italics*, without capital letter. All words in section and chapter headings, will be typeset using the style that is defined for the specific heading.

Well known programming languages like Matlab® and other trademarks like Windows® will be denoted with capital letter, as names, and will not be emphasized further, other than receiving the trademark symbol or the registered trademark symbol the first time they are mentioned. All trademarks and registered trademarks mentioned in this text are properties of their respective owners.

This document was typeset using the LaTeX version of the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography, *The Elements of Typographic Style*.

ABSTRACT

Nonlinear wave propagation has, for the last couple of decades, become an increasingly more important tool in medical ultrasound imaging. Creating ultrasound images from echoes in the second harmonic frequency band provides a major enhancement to the image quality, reducing body wall reverberation and also reducing perturbations from off-axis echoes.

The aim of this study has been to make a fast implementation of nonlinear wave simulation in ultrasound, based on the *Propose* method, and investigate its agreement with the state-of-the art simulation tool, *Abersim*. Agreement between the methods has been investigated for a typical cardiac probe in the fundamental and second harmonic frequency band. The comparison showed good agreement for phase, amplitude, wave form and beam profile. Some overestimation of the center lobe in the beam profile was found, but this is expected for quasilinear methods.

A fast implementation for simulation of nonlinear wave propagation in ultrasound has been made, based on the *Propose* method [2]. The new implementation and is performing simulations in the second harmonic frequency band 74% – 93% faster than the original implementation.

SAMMENDRAG

Ikkelineær bølgeforplanting har i de siste par tiår spilt en stadig viktigere rolle innen medisinsk ultralydavbildning. Ved å generere bilder fra ekko i det andreharmoniske frekvensbåndet, fremfor fundamentalbåndet, oppnås en forbedret bildekvalitet. Hovedårsaken til dette er redusert reverberasjonstøy fra overflatelagene i medisinsk vev.

Målet med denne masteroppgaven har vært å implementere en rask simulator for ikkelineær bølgeforplanting i ultralyd, basert på *Propose* metoden, og deretter undersøke graden av samsvar mot det anerkjente simuleringsverktøyet Abersim.

Sammenligningen av en rektangulær hjertetransducers ultralydfelt i fundamentalbåndet og andre harmoniske frekvensbånd viste godt samsvar for hovedloben og de sentrale delene av feltet, både når det gjelder fase, amplitude bølgeprofil og pulsform. Noe overestimering av effekten i senterloben ble observert, men dette er helt å forvente når man bruker kvasilineær teori.

En rask implementasjon for simulering av ikkelinær bølgeforplanting i ultralyd har blitt laget, basert på Propose metoden [2]. Den nye utgaven utfører nå simuleringer i det andre harmoniske frekvensbånd med 74% – 93% redusert kjøretid, sammenlignet med den opprinnelige versjonen.

CONTENTS

ACRONYMS

CPU	Central Processing Unit
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FT	Fourier Transform
GCC	GNU Compiler Collection
GPU	Graphical Processing Unit
IFFT	Inverse Fast Fourier Transform
IFT	Inverse Fourier Transform
KZK	Khokhlov-Zabolotskaya-Kuznetsov
MEX	Matlab Executable
MI	Mechanical Index
OpenMP	Open Multi-Processing
OS	Operating System
PDE	Partial Differential Equation
RAM	Random Access Memory
RMS	Root Mean Square
THI	Tissue Harmonic Imaging

LISTINGS

1 INTRODUCTION

A sound pulse transmitted through a nonlinear medium, such as water, will propagate with pressure dependent local velocity. This causes deformations in the propagating pulse because the local pressure is different in various parts of the sound wave. Peak compressions will tend to catch up on the peak rarefactions resulting in a sawtooth shaping [3, 4] of the original sine wave. In the frequency domain, this effect is visible as the generation of higher order frequency modes. Effects and applications of nonlinearity in medical ultrasound are explained and illustrated thoroughly in Duck's review article on the subject [4].

During the last two decades, nonlinear wave propagation has become an increasingly important key for enhancing image quality in medical ultrasound imaging. In Tissue Harmonic Imaging (THI), echoes in the second harmonic frequency band are used to construct the ultrasound picture instead of those at fundamental frequency.

A reason for the enhanced image quality is that ultrasound imaging in the fundamental mode contains multiple reflections from the transition surface between the transducer and layers in the body wall. These echoes will generate reverberation noise, obscuring the imaging of tissue located inside the body. For THI such surface noise will be avoided, because the second harmonic waves are generated from the fundamental pulse during propagation. Therefore waves with imaging frequency are not present in the initial transition through the body wall [4, 5]. Dense boundary surfaces inside the body of a patient may however reduce the enhancing effect [6]. The reason for this being that the second harmonic waves may have reached amplitudes where they are no longer negligible.

THI is documented to improve diagnostic image quality in general [7], and also in many specific applications. Concrete examples of this are obstetrics and abdominal imaging [8], measurement of heart functions [8, 9, 10], examination of liver [8, 11, 12] and examination of kidneys [8, 13]. The use of THI is documented to provide better means to distinguish structure and clutter inside fluid filled organs such as the gall bladder and the pregnant uterus [8, 14]. This is explained by improved ability to define edges of structures. Improved recognition of edges is also useful in endocardial border definition [8, 15, 16].

An important prerequisite for obtaining high quality ultrasound pictures, is the ability to predict the magnitude and spatial distribution of the transmitted pulse[27]. Because of this, a large effort has been made in the international research community to develop efficient ultrasound simulation tools. The different methods have slightly varying approaches to the same problem, which is to make accurate and fast prediction of nonlinear ultrasound fields. Unfortunately, fast and accurate is not two qualities which necessarily fit together, and compromises often need to be made in the favor of one of them. Mathematically the approaches may perform calculations in the time domain, in the Fourier domain or partly in both [17].

Examples of simulation tools where the accuracy is high, is the time domain method called *KZK-Texas* [18], the Fourier domain method *KZK-Bergen* [19, 20] and the *Abersim* method [21, 22], which performs calculations partly in both domains.

In the Fourier domain, multiple implementations has been made [2, 23, 24] which relies on a quasilinear approximation [25]. In these methods, computational speed has been prioritized at the expense of some accuracy. However, when the nonlinear effects are small, they provide sufficiently accurate results for many applications.

The quasilinear simulation method, *Propose*, has earlier been implemented using Matlab® [2, 26, 27, 28]. The aim of the present thesis is to make a fast implementation of nonlinear wave simulation in ultrasound, based on the *Propose* method, and investigate its agreement with the state-of-the art simulation tool, *Abersim*. Agreement between the methods are investigated for a typical cardiac probe, with respect to both magnitude and phase, at multiple depth ranges.

OUTLINE OF THIS THESIS

Chapter 2 in thesis opens with a short review of the theoretical fundament in different approaches to simulations of nonlinear wave propagation in ultrasound. Especially the quasilinear approach is emphasized. Further, a general wave equation is presented, from which the rest of the mathematical theory is developed. Governing equations in both *Abersim* and *Propose* will be presented thoroughly. at the end of the chapter, some results from my specialization project [1] and theory about the Mechanical Index (MI) are presented.

The methods are discussed in Chapter 3. First, the computers used in the study are presented, followed by setup parameters used for comparisons of *Propose* and *Abersim*. The comparison study itself will also be presented, followed by an outline of the methods used to determine how fast the presented implementation of *Propose* performs simulations.

In Chapter 4 the results are presented, starting with the comparable figures of beam profiles and field plots and at multiple depth ranges. In the end of the chapter the computational speed of the new implementation are presented, in a way which enables comparison to earlier presented methods.

The discussion, presented in Chapter 5, follows a pattern similar to the one in Chapter 4. First the accordance between *Propose*, and the state-of-the-art simulation tool, *Abersim* is discussed, and related to previously presented studies. Secondly the speed of the new implementation is discussed. Multiple interesting elements regarding generality of the results and performance in different Operating System (OS)es is also emphasized. This chapter closes with a discussion on possible errors in the study and possible drawbacks in the *Propose* method.

The final conclusion and objectives for future studies, are presented in Chapter 6.

In Appendix A, a preliminary study of new findings postponing the results in my specialization project [1] is presented. The specific issues are also presented in Section 2.7.

Appendix B presents a second preliminary study. This study is included to justify some optimization choices made in Section 3.3, and statements about the matter made in Section 2.6.4.

Finally, the source code for all the C++ implementations are included in Appendix C. Initial versions of the MEX functions in Listing 4 and 5 have previously been presented as part of my specialization project [1]. Preceding each listing is a note on updates made as part of the current study.

2 THEORY

2.1 REVIEW OF NONLINEAR ULTRASOUND SIMULATION

To obtain optimal quality in ultrasound images, the ability to predict amplitude level and the spatial distribution of the transmitted wave is of high importance [2]. Therefore, a large number of numerical simulation methods have been developed to model nonlinear wave propagation. According to [17], the various algorithms may be divided into three main categories: Calculations performed in the time/space domain, in the frequency domain or a combination of these. Important model equations used in numerical implementations are derived from fundamental principles in [29]. The nonlinear wave equation most widely used is the Khokhlov-Zabolotskaya-Kuznetsov (KZK) equation [30, 31], from which many methods both in time domain and the Fourier domain draw their foundational concepts.

Direct time domain approaches rely on finite difference methods to make approximate solutions regarding every aspect of the governing equations. In this approach the waveform may be arbitrarily chosen because all frequency components are simultaneously accounted for in the time domain. The drawback of the time domain algorithms is that they require stepwise propagation from the source to the point of interest, and thus will require increasingly more computational time as the propagation distance is increased. The computational time needed to include nonlinearity in a time domain method is proportional to the number of points N in the time domain waveform, regardless of how many frequency components the signal consists of [18].

In the frequency domain approach it is assumed that a finite pulse may be written as a superposition of multiple periodical signals. This makes it possible to express the pulse by its frequency components in the Fourer domain. By calculating wave propagation in the Fourier domain one is able to simulate the wave field in a desired plane directly by applying a phase shift in the Fourier domain, instead of performing stepwise propagation. This is known as the angular spectrum method, and is described mathematically in Section 2.3.

Computationally efficient description of a system in the Fourier domain requires source waveforms which can be described by a limited number of frequency components. The computational time in a spectral method is proportional to M^2, where M is the number of frequency components included in the Fourier domain description of the pulse. The M^2 dependence in the frequency domain is related to the nonlinear part of the frequency domain KZK equation. The absorption and diffraction is solved in a speed proportional to M [18].

2.1.1 *General methods*

The best known time domain implementation is the KZK-Texas code [18] which uses the finite difference method and the method of fractional steps to simulate an axisymmetric system term by term.

The first numerical implementation of the KZK equation in the frequency domain is the KZK-Bergen code [19, 20]. This method was created for simulation of the near field radiated from axisymmetric sources. The method used two coupled sets of equations to determine the Fourier series coefficients and did not use an expanding coordinate grid, resulting in high computational time for the far field. An improved version of the method was later presented in [32]. Details of the KZK-Bergen method are outlined briefly in [25].

A problem with the KZK equation is that it depends on a parabolic approximation, assuming that the acoustic energy is propagating in a narrow beam. This is not true when the source is strongly focused, its dimensions approaches one wavelength or when the observation point is close to the source or far off the axis. Therefore, algorithms using the equation often assume directional sound beams.

To simulate more general problems, a frequency domain method that does not rely on the parabolic approximation has been presented in [33]. This method introduces diffraction using an exact Kirchhoff-Helmholtz integral [17] which is solved using a discrete Hankel transform [34]. As it performs calculations in the frequency domain it is limited by the M^2 dependency on the number of frequency components M in the signal.

A well established hybrid method is the *Abersim* algorithm [21, 22] which has been used as a reference tool in this thesis and is presented in detail in Section 2.4. Other hybrid methods are presented in [35, 36]. In these combined methods, the diffraction and absorption part of the KZK equation is solved in the frequency domain, and the effects of nonlinearity is handled in the time domain. All the physical processes in the wave propagation are regarded as independent, allowing for calculations of each process in the domain that gives highest benefits in regard to speed and accuracy, before combining them to a final solution using superpositioning. These methods apply a Fast Fourier Transform (FFT) which requires a computational time proportional to $M \log M$ [18].

2.1.2 *Quasilinear methods*

If a system can be sufficiently described using the fundamental and the second harmonic frequency mode, the quasilinear approximation [25] may be deployed to ease the computational load of frequency domain approaches (Section 2.5.2). The simulations employing the quasilinear approximation avoid the earlier mentioned M^2 computational time dependency, because the fundamental field is regarded as an unaffected source term of the second harmonic field (Section 2.5.3).

One algorithm utilizing this approximation is presented in [24]. The method uses a generalized angular spectrum method (Section 2.3) to simulate a second harmonic signal in a heterogeneous medium. Because the coefficient of nonlinearity β [37] varies in a hetrogeneous medium, the method rely on stepwise propagation in resemblance with time domain methods.

A nonlinear extension to the Field II [38] simulator is presented in [23] utilizing quasilinear theory. This method neglect interactions between temporal frequency components, which introduces limitations to the bandwidth for which the method is valid.

A method of special interest is the *Propose* method [2, 26, 27] treated more thoroughly in Section 2.5. This method is the basis for the fast implementation that will be presented in this thesis. The method rely on the quasilinear approximation (Section 2.5.2) and assumes homogeneous medium, ensuring a constant coefficient of nonlinearity β. Diffraction is solved using the angular spectrum method (Section 2.3) over arbitrary distances, allowing for direct simulation of the pulse amplitude at any depth. The objective of the *Propose* method is to be fast enough to allow seamless automatic adjustment of setup parameters in a medical scanner, whenever a user alters the imaging parameters [2]. The method is also suggested as a possible everyday tool for transducer designers [27].

The results obtained in [27] has been further exploited in [39], where the quasilinear *Propose* method was compared to water tank measurements as well as to an early version of the *Abersim* simulation software [40]. The comparisons were done for annular symmetry and it was found that the quasilinear approximation gives reasonable results when the MI (Section 2.8) is small. It was found that when the MI becomes higher than approximately 0.5, side lobe levels will be underestimated in the *Propose* method. This is because the higher order harmonics no longer will be negligible, as is the premiss for using a quasilinear approach (Section 2.5.2).

In [26], further verification was carried out, this time both the annular array probe and a rectangular probe was compared to measurements and the early *Abersim* version [40]. It was found that the second harmonic magnitudes are underestimated by less than 0.1 MPa for MI below 0.4 in the axial symmetrical case, and below 1 for the rectangular probe. In the present thesis, the simulated field from an identically shaped rectangular transducer will be explored and compared to simulations from the current *Abersim* version [22, 21].

2.2 NONLINEAR ULTRASOUND WAVE EQUATION

Nonlinear wave propagation can for sound in an absorbing, homogeneous fluid without dispersion, generally be described by the second order Partial Differential Equation (PDE) [2]:

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = -\frac{\beta\kappa}{c^2}\frac{\partial^2 p^2}{\partial t^2} + \Lambda(p)\,. \tag{1}$$

In this equation, ∇^2 is the Laplace operator denoting partial differentiation in all spatial directions, c is the local speed of sound, p is the acoustical pressure and $\Lambda(p)$ is an operator accounting for losses. The density dependent coefficient of compressibility is denoted as $\kappa = 1/\rho c^2$ and $\beta = 1 - B/2A$ is the coefficient of nonlinearity defined by the Taylor coefficients A and B in the pressure-density relation addressed further in [37]. In medical ultrasound the variations of the phase velocity with frequency is very small and therefore the dispersion is usually neglected in simulations [41, 2].

A generalized Westervelt equation [42, 29], assuming plane waves and neglecting dispersion [43, 22, 44] can be obtained by setting the loss operator $\Lambda(p) = 1/c^2 \cdot \partial^2 \mathcal{L} p / \partial t^2$:

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = -\frac{\beta\kappa}{c^2}\frac{\partial^2 p^2}{\partial t^2} - \frac{1}{c^2}\frac{\partial^2 \mathcal{L} p}{\partial t^2}. \tag{2}$$

In this equation, $\mathcal{L}$ can be viewed as the kernel of a convolution operator describing attenuation properties (28) [44, pp. 40 & 44]. This equation is true for homogeneous media, and accounts for both diffraction and attenuation.

In Abersim, heterogeneities can be included in the simulation. This is done by introducing a delay screen body wall to impose a phase shift between propagation steps. This way the density between to screens may be regarded constant, and the homogeneous propagation equation may be used [22, 44]. To be able to use the *Propose* method, homogeneous media is required. Hence, all simulations presented in this thesis will picture ultrasound pulsed propagating in distilled water.

To transform (2) into a directional equation, the retarded time variable

$$\tau = t - z/c, \tag{3}$$

is introduced. For directional sound beams the parabolic approximation[1] is valid [22, 45] and we get $\nabla \approx \nabla_\perp^2$. This will lead to the PDE:

$$\frac{\partial^2 p}{\partial \tau \partial z} = \frac{1}{2}\nabla_\perp^2 p - \epsilon_t \frac{\partial^2 p}{\partial \tau^2} + \frac{\epsilon_n}{2}\frac{\partial^2 p^2}{\partial \tau^2} + \frac{1}{2c^2}\frac{\partial^2 \mathcal{L} p}{\partial \tau^2}, \tag{4}$$

where $\nabla_\perp^2 = \nabla^2 - \partial^2/\partial z^2$. The scaling variables ϵ_n and ϵ_t will depend on the medium and its temperature and are constant in regard to τ and z. Specific values for different types of biological tissue are given in [22, p. 186].

2.3 THE ANGULAR SPECTRUM METHOD – AN OVERVIEW

The angular spectrum method is used for calculating the diffraction in both *Abersim* and *Propose*. The difference between the two methods is mainly in the step length h. While *Propose* performs simulation of the field at an arbitrary depth z in one single step. *Abersim* will obtain the wave field at the same depth by

1 The parabolic approximation leads to: $\partial^2 p / \partial z^2 \approx 0$.

a series of small steps according to (14). The articles regarding the two methods have slightly different approaches, depending on whether or not a retarded time variable is used in the equations. Therefor both sets of equations will be developed in the in Sections 2.4.2 and 2.5.3, for *Abersim* and *Propose* respectively

In this section the general idea of the method is presented to provide a general overview of the concepts. This section is based on discussions in [47, 46].

Consider a source free half space where a pulse, p, with frequency f_0, is transmitted. Further, assume that this pulse can be decomposed into a set of monochromatic plane waves with various wave numbers in the three dimensions x, y and t:

$$p(x, y, z, t) = \iiint P(x, y, z, t)\, e^{i(\omega t + k_x x + k_y y)} dt dx dy. \qquad (5)$$

The angular spectrum for the given depth z, denoted by P, is a three dimensional matrix containing information about all the amplitudes related to the various frequency components.

In (5) one can recognize the three dimensional Fourier Transform (FT) in time and the spatial dimensions x and y. Hence, the angular spectrum $P(\vec{k}, z)$ for a plane z may be computed from its corresponding space-time sound pressure field using an Inverse Fourier Transform (IFT):

$$P\left(\vec{k}, z\right) = \iiint p(x, y, z, t)\, e^{-i(ck_t t + k_x x + k_y y)} dt dx dy, \qquad (6)$$

where

$$\vec{k} = [k_t, k_x, k_y] \qquad (7)$$

is a vector containing the three wave numbers, $k_t = \frac{\omega}{c}$ and ω denotes the angular frequency of the signal.

By knowing the FT of the sound field at a given depth, one may calculate the field at any other depth by multiplying with a phase factor in the Fourier domain:

$$P\left(\vec{k}, z\right) = P(x, y, z_0, t)\, e^{-i\kappa(\vec{k})(z - z_0)}. \qquad (8)$$

This is the angular spectrum method. The variable $\kappa(\vec{k})$ is a complex three dimensional wave number operator which depends on the governing wave equations. It will also depend on whether or not retarded time is used and whether or not any loss operators have been included. This operator will be developed in the following for both *Abersim* (22) and *Propose* (37).

As shown in (5), a known angular spectrum (8) is easily used to obtain the pressure field in time and space, using an IFT.

2.4 THE ABERSIM METHOD

The *Abersim* simulation software has been verified through multiple articles the last decade [21, 22, 39, 40] and is designed to simulate directional wave propagation in retarded time, using a sine signal multiplied with a Gaussian envelope function. The software is able to simulate waves in both forward propagation and back scattering, however not at the same time. Back scattering is modeled from the known field of a forward propagating wave, using either a small group of point scatters, or a single one. Because back scattering is omitted in the *Propose* method (Section 2.5), it is not discussed further in the present thesis.

Effects of nonlinearity and attenuation is built into the underlying, directional wave equation (4), which the Abersim method is solved using an operator splitting approach [22].

Abersim is written in Matlab under the GNU Public License [50] and can be downloaded from the project web page [51]. The *Abersim* download package contain several tutorials to help new users get acquainted with the software. There is also included a detailed manual describing the software [44]. A user may choose to instal an included C version or a MEX accelerated Matlab version but none of these will compile on the GNU Compiler Collection (GCC) 4.8 or 4.9 at the time of writing. Hence, the Matlab-only version of is used in this thesis. Using C or MEX versions would only affect the computational speed, and not the simulation output.

2.4.1 *Governing equation*

The directional and dimensionless plane wave equation which is used in *Abersim* is obtained by integration of (4) over all history in retarded time [22]:

$$\frac{\partial p}{\partial z} = \frac{1}{2}\int \nabla_\perp^2 p \, d\tau + (\epsilon_n p - \epsilon_t)\frac{\partial p}{\partial \tau} + \epsilon \frac{\partial L p}{\partial \tau}. \tag{9}$$

Here, the first term accounts for diffraction, the second for nonlinearity and the third for attenuation. Introduction of the tissue dependent variable[2] ϵ provides a change from $\mathcal{L}$ to L, defined by $\epsilon L = \mathcal{L}/2c^2$.

2 If heterogeneous media were considered, ϵ would also depend on z.

2.4.2 *Operator splitting*

The operator splitting approach outlined in [22], is based on the Lie-Trotter product formula [52] and the product integral [53]. Operator splitting can be performed when each term on the right hand side of (9) yield independent solutions. Leaning on corresponding sections in [22] and [44], this section will briefly present each of the three terms.

Equation (9) is in the form

$$\frac{\partial p}{\partial z} = (A_d + A_n + A_a)p, \tag{10}$$

with A_d, A_n, A_a accounting for diffraction, nonlinearity and attenuation respectively. Following this notation we may parse (9) into three independent parts:

$$A_d p = \frac{1}{2}\int \nabla_\perp^2 p \, d\tau, \tag{11}$$

$$A_n p = (\epsilon_n p - \epsilon_t)\frac{\partial p}{\partial \tau}, \tag{12}$$

$$A_a p = \epsilon \frac{\partial \mathcal{L} p}{\partial \tau}, \tag{13}$$

where ϵ is a tissue dependent constant (27).

A forward propagating wave is calculated in *Abersim* using an evolution equation which is performing a propagation in steps of length h in the direction of z. The operator which performs the transition from an initial state $\psi(0)$ to the next state $\psi(h)$ is described by the exponential function $\exp\left(h \sum_i A_i\right)$ where the sum includes all A_i terms of (10):

$$p(z_{k+1}) \approx e^{(h\sum_i A_i)} p(z_k). \tag{14}$$

The z-axis position at a given step number k, is expressed as $z_k = kh$, and the evolutionary solution from step k, to step $k+1$. The approximation in (14) is of first order and has a formal error of order $\mathcal{O}(h^2)$. This method is called Gudonov splitting [54], and according to [22] an alternative model called Stang splitting [55] may also be used.

Implementation in Abersim

Abersim models the forward propagating wave using an evolution equation. Starting with the initial condition

$$p(x, y, 0, t) = p_0(x, y, t),$$

in $z = 0$ the wave is propagated in steps of length h, in the positive z-direction [22]. The total numerical solution may be denoted as

$$p(z_{k+1}, t) = \Psi^h_{A_d}(z_k) \cdot \Psi^h_{A_n}(z_k) \cdot \Psi^h_{A_a}(z_k) \cdot p(z_k, t), \tag{15}$$

where $\Psi^h_{A_i}$ denotes the numerical approximation of $\exp(hA_i)$ from (14).

Diffraction term – angular spectrum method in retarded time

The diffraction in Abersim is accounted for using the angular spectrum method (Section 2.3) in small steps h.

From the Westervelt equation (4), the diffraction term may be recognized as the linear wave equation for a lossless, homogenous medium [44]:

$$\frac{\partial^2 p}{\partial \tau \partial z} = \frac{1}{2} \nabla_\perp^2 p. \tag{16}$$

Extracting the term in ∇^2 that depends on the z direction, we obtain a PDE which is coupled in x, y and τ:

$$\frac{\partial^2 p}{\partial z^2} - 2\frac{\partial^2 p}{\partial \tau \partial z} + \nabla_\perp^2 p = 0. \tag{17}$$

By performing a three dimensional FT[3] in retarded time τ and the two spatial directions x and y, we get a decoupling in space and time:

$$\frac{\partial^2 P}{\partial z^2} - 2ik_\tau \frac{\partial P}{\partial z} - (k_x^2 + k_y^2)P = 0. \tag{18}$$

In (18), P denotes the FT of p in retarded time τ and the spatial dimensions x and y. Note that the direction of propagation, z

3 The FT properties used to obtain (18):

$$\mathcal{F}\left\{\frac{\partial^n p}{dt^n}\right\} = (i\omega)^n \mathcal{F}\{p\} \quad \text{and} \quad \mathcal{F}\left\{\frac{\partial^n p}{ds^n}\right\} = (ik_s)^n \mathcal{F}\{p\}$$

where $\omega = ck_\tau$ is the angular frequency in the temporal domain and s denotes a spatial direction (here: x and y) [48].

is left out of the FT. The variables k_x and k_y denotes the spatial wave numbers in the x and y directions, expressed in m^{-1}. The variable, $k_\tau = \omega/c$, may be viewed as a wave number in retarded time , which is expressed in m^{-1}.

A solution to (18) will be on the form [22]:

$$P(z_o + h) = Ae^{i(\sqrt{k_\tau^2 - k_x^2 - k_y^2} - k_\tau))h} + Be^{-i(\sqrt{k_\tau^2 - k_x^2 - k_y^2} - k_\tau)h}, \quad (19)$$

where the first wave propagates in positive z-direction, and the second one propagates in negative z-direction. Ignoring the solution that propagates backwards, the solution becomes

$$p(z_o + h) = \mathcal{F}^{-1}\left\{P(z_0) \cdot e^{-i\mathcal{K}_z h}\right\}, \quad (20)$$

where, $\mathcal{F}^{-1}$ denotes the IFT in x, y and τ and we have defined the scalar, frequency domain wave number operator:

$$\mathcal{K}_z = \sqrt{k_\tau^2 - k_x^2 - k_y^2} - k_\tau. \quad (21)$$

Equations (20) and (21) corresponds to the conceptual angular spectrum equation (8) presented in Section 2.3.

Very close to the transducer, the transversal wave numbers, k_x and k_y, might be large enough for (21) to become imaginary. This is caused by the presence of evanescent waves which are quickly attenuated [49], but the solution has to be treated with special care to avoid rising the exponent term in (20) by a positive real power.

In *Abersim*, the evanescent waves are suppressed in (20) by the following modification of the $\mathcal{K}_z$ operator in (21)[4]:

$$\mathcal{K}_z = \begin{cases} \sqrt{k_\tau^2 - k_x^2 - k_y^2} - k_\tau & \text{for } k_\tau^2 - k_x^2 - k_y^2 > 0 \\ -\sqrt{k_\tau^2 - k_x^2 - k_y^2} + k_\tau & \text{otherwise.} \end{cases} \quad (22)$$

This multiplication by -1 is equivalent to picking the backwards propagating solution in (19). The modification ensures that the solution (20) does not diverge, but are quickly suppressed as the exponent now contains a negative real term [22].

From (20) the diffraction term of (15) may be recognized as:

$$\Psi^h_{A_d}(z) = \mathcal{F}^{-1}\left\{e^{-i\mathcal{K}_z h}\right\}. \quad (23)$$

4 This problem has been approached a little differently in *Propose* (Section 2.5.3).

Nonlinearity term

Equation (12) governes the nonlinearity properties in *Abersim*. For small step sizes h, a solution is found by the *method of characteristics* [56]:

$$p(z_{k+1}, \tau) = p\left(z_k, \tau_k - h \cdot \Delta\left[z_k, p(z_k, \tau_k)\right]\right) \tag{24}$$

with

$$\Delta\left[z_k, p(z_k, \tau_k)\right] = p(z_k, \tau_k) \cdot \frac{\epsilon_n(z_k) + \epsilon_n(z_{k+1})}{2} - \frac{\epsilon_t(z_k) + \epsilon_t(z_k + 1)}{2}.$$

In (24), the grid points may experience a perturbation which depends on the local pressure. This in turn will cause the points to become in-equally spaced in time. To compensate, the pressure is re-sampled with equal time intervals introducing an interpolation error. This error will for sufficiently high sampling frequency become negligible. The step size and sampling frequency are adapted at runtime in *Abersim* to avoid shock formation [22]. The solution operator (24), including the re-sampling routine, defines the nonlinear term $\Psi^h_{A_n}(z)$ in (15).

Attenuation term

The attenuation term presented in (13) may for ultrasound in soft tissue be modeled using a frequency dependent power law of the form

$$\alpha(f) \propto af^b, \tag{25}$$

where a and b are tissue dependent parameters [57, p. 112]. With this model, the right hand side of (13) is defined by its temporal FT [22]:

$$\mathcal{F}\left\{\frac{\partial Lp}{\partial \tau}\right\} = -|\omega|^b \mathcal{F}_\tau\{p\}, \tag{26}$$

and the tissue dependent variable

$$\epsilon = \frac{\ln 10}{20} \frac{a}{(2\pi)^b}. \tag{27}$$

Equation (26) does not obey causality and therefore a modified definition is used in Abersim:

$$\mathcal{F}\left\{\frac{\partial Lp}{\partial \tau}\right\} = \left(-|\omega|^b + i\mathcal{H}\left\{|\omega|^b\right\}\right) \mathcal{F}_\tau\{p\}, \tag{28}$$

where $\mathcal{H}$ denotes the Hilbert transform [58] and i is the imaginary unity vector. From this definition the attenuation contribution to (15) is defined as

$$\Psi^{h}_{A_a}(z) = \mathcal{F}^{-1}\left[e^{\epsilon\cdot\left(-|\omega|^{b_z}+i\mathcal{H}\{|\omega|^{b_z}\}\right)}\right]. \tag{29}$$

For simulations and comparison of methods in this study we have omitted attenuation, simulating propagation in distilled water.

2.5 THE *propose* METHOD

2.5.1 *The Matlab implementation of* Propose *[2]*

A Matlab implementation of the *Propose* method has been developed as part of the work in [2] and [27]. An overview and explanation of the work flow in the program, and various subroutines are given in my specialization project [1]. The optimization work in the present thesis has been done on the subroutine *prop2harm*, which calculates the second harmonic field.

2.5.2 *The quasilinear approximation*

A foundational concept in the *Propose* method is to simulate propagation in the second harmonic frequency band using the quasilinear approximation [25]. In this theory it is assumed that wave modes at higher order frequencies are generated without perturbating the fundamental field. This is approximately true when the fundamental signal contains much more energy than the second harmonic, $p_1 >> p_2$. With this premise the fundamental signal p_1 may first be calculated independently. Knowing the fundamental signal, the second harmonic signal p_2 is found, using the the fundamental signal as part of its source term. The total pressure p, will afterwords be found by superimposing the two contributions [59]:

$$p = p_1 + p_2. \tag{30}$$

The quasilinear approximation is only applicable up to the second harmonic frequency band, hence we neglect contributions of higher order than $\mathcal{O}(2)$.

Calculations in the *Propose* method are performed in the Fourier space using the angular spectrum approach (Sec 2.5.3). The method allows for calculation of the sound field in any plane z, from a known reference plane z_0 without stepwise propagation between the two states.

2.5.3 *Mathematical foundation*

In this section, the mathematical theory of the *Propose* simulation method is presented. The outline is based on corresponding sections in [2] and [27]. We start out with the assumptions and the nonlinear wave equation (1) presented in Section 2

Using the quasilinear approximation (30) we may prepare two independent versions of (1), one with the signal pressure in the fundamental frequency band

$$\nabla^2 p_1 - \frac{1}{c^2}\frac{\partial^2 p_1}{\partial t^2} + \Lambda(p_1) = 0, \tag{31}$$

and one determining the second harmonic field

$$\nabla^2 p_2 - \frac{1}{c^2}\frac{\partial^2 p_2}{\partial t^2} + \Lambda(p_2) = -\frac{\beta\kappa}{c^2}\frac{\partial^2 p_1^2}{\partial t^2}. \tag{32}$$

The right hand side of (32) can be interpreted as a source term for the second harmonic field p_2 which depends on the fundamental field p_1.

The angular spectrum approach in Propose

The diffraction in propose is solved using the angular spectrum method, presented in Section 2.3. The simulation is performed in a single step from a reference depth z_0 to the depth of interest z, without stepwise propagation.

Using properties of the FT, the Eqs. (31) and (32) may be expressed in the Fourier domain as

$$\frac{\partial^2 P_1\left(\vec{k},z\right)}{\partial z^2} + K^2(\vec{k})P_1\left(\vec{k},z\right) = 0 \tag{33}$$

and

$$\frac{\partial^2 P_2\left(\vec{k},z\right)}{\partial z^2} + K^2(\vec{k})P_2\left(\vec{k},z\right) = \frac{\beta\kappa\omega^2}{2c^2}P_1\left(\vec{k},z\right) \otimes P_1\left(\vec{k},z\right). \tag{34}$$

The symbol $\otimes$ represents a convolution over all three dimensions in $\vec{k}$, P_1 and P_2 is the angular spectrum for the fundamental and second harmonic signal pressure respectively. K is a scalar three dimensional wave number operator

$$K\left(\vec{k}\right) = \sqrt{k_t^2 - k_x^2 - k_y^2}, \tag{35}$$

where $k_t = \omega/c$ can be viewed as a time domain wave number. The equations (33) and (34) are solved in the Fourier domain, and the space-time pressure distrbutions p_1 and p_2 are obtained from their angular spectrum using FFT to solve (5).

As mentioned for the *Abersim* case (Section 2.4.2), evanescent waves, may cause (35) to become imaginary. In the original Matlab implementation [2], evanescent waves are suppressed by ignoring all imaginary values of K:

$$p(z_o + h) = \begin{cases} \mathcal{F}^{-1}\left\{P(z_0) \cdot e^{-i\mathcal{K}_z h}\right\} & \text{when } \mathcal{K}_z \text{ is real} \\ \mathcal{F}^{-1}\{0\} & \text{othervise.} \end{cases} \tag{36}$$

This approach is relaying on the fact that the evanescent waves are quickly attenuated and thus represent a small contribution to the total field, which can be neglected without loss of precision [2, 49].

In the new implementation of *Propose* (Listing 5), the more precise (22) has been used in stead of (36). The reason for this is that (22) does not represent a computationally more demanding process in the C++ implementation. All data points in the field are included in the for-loop, and there are little time gained by using the simplified version. The structure of the Matlab implementation benefits more from using (36), because some values in the field might be excluded from the main Fourier domain computations.

Equation (37) is in accordance with the corresponding equation in [2] and [27], and the reason why it does not correspond directly to the *Abersim* version (21) is that retarded time has not been taken into account in the *Propose* articles. However, the expression used for $K\left(\vec{k}\right)$ in the original Matlab implementation [2], corresponds to the version used in *Abersim* (21).

It is fortunate to use retarded time in the *Propose* method as well as in *Abersim*, because this will ensure that the simulated field is placed in the center of the output matrix. The Fourier domain algorithm repletes the simulated field periodically in all directions and dismissing retarded time will allow the grid of periodic repetitions to move around in the matrix, and hence also in the output plots. Use of retarded time will avoid this, therefor (22) has been used in the new MEX implementation, according to the original Matlab version.

A retarded time variable $\tau = t - z/c$ in *Propose* can, analogue to the *Abersim* approach (Section 2.4.2), be inferred before the FT of (31) and (32), leading to a parallel reasoning, which will result in an expression identical to (21). The rest of the *Propose* theory will be unaffected by either choice of K.

Attenuation

Attenuation is accounted for in *Propose* by appending a negative imaginary part to (35) leading to [2]:

$$K\left(\vec{k}\right) = \sqrt{k_t^2 - k_x^2 - k_y^2} - ia\left(\frac{f}{10^6}\right)^b, \tag{37}$$

where f denotes frequency and a and b are tissue dependent parameters governing the power law attenuation (25) in biological media. The appended expression can be interpreted as the frequency domain version of the attenuation operator Λ in (1).

Combining (26) and (27) and knowing that $\omega = 2\pi f$. One may discover that the attenuation term appended in (37), only differs from the *Abersim* version (13) by a frequency independent factor.

As mentioned earlier, lossless medium is assumed in this thesis by setting a and b to zero.

Solutions used in the Propose *method*

A solution to (33) is

$$P_1\left(\vec{k}, z\right) = P_1\left(\vec{k}, z_0\right) e^{-iK(\vec{k})(z-z_0)}, \tag{38}$$

where the sign of the exponent is chosen in conjunction with the sign of the attenuation term in K (37) to prevent the solution from diverging when $z \to \infty$. Note that this equation corresponds to the conceptual angular spectrum equation (8) presented in Section 2.3.

The second harmonic field is governed by (34) and consists of a particular solution $P_{2p}\left(\vec{k}, z\right)$ and the homogeneous solution $P_{2h}\left(\vec{k}, z\right)$. The particular solution is developed in [2] using the Green's function for a half space defined by the source plane and the direction of propagation. Any secondary waves, which have been reflected once in the source plane z_0 before it reaches observation depth z, is neglected. Contributions from sources beyond the observation plane propagating backwards are also neglected. The resulting solution describe a forward propagating wave, created by sources between the transducer z_0 and the observation depth z, which represents the dominating second harmonic contribution [2, 60]:

$$P_2\left(\vec{k},z\right) \approx \frac{iM}{2K(\vec{k})}\int_0^z \left\{P_1\left(\vec{k},z'\right) \otimes P_1\left(\vec{k},z'\right)\right\} e^{-iK(\vec{k})(z-z')}dz', \tag{39}$$

where $M = (\beta\kappa\omega^2)/(2c^2)$ and $P_1\left(\vec{k},z'\right)$ is the angular spectrum of the fundamental field given by (38).

At the transducer surface $z = 0$, we assume the total second harmonic field to be zero

$$P_2\left(\vec{k},0\right) = P_{2p}\left(\vec{k},0\right) + P_{2h}\left(\vec{k},0\right) = 0.$$

Note that the integral in (39) becomes zero in $z = 0$ as well. Therefore, the homogeneous solution has to be zero also, and the total second harmonic field is described by its particular solution:

$$P_2 = P_{2p}.$$

Using this result, and inserting the expression for P_1 (38) into (39) the following expression for the second harmonic field is obtained:

$$P_2\left(\vec{k},z\right) = \frac{iM}{2K(\vec{k})}\iiint P_1\left(\vec{k}',z_0\right)\cdot P_1\left(\vec{k}-\vec{k}',z_0\right)\cdot Hp\left(\vec{k},\vec{k}',z,z_0\right)\frac{d\vec{k}'}{(2\pi)^3}, \tag{40}$$

where

$$Hp\left(\vec{k},\vec{k}',z,z_0\right) = ze^{-iK(\vec{k})(z-z_0)}\cdot e^{-i\gamma(\vec{k},\vec{k}')(z_0-z/2)}\cdot \mathrm{sinc}\left(\gamma(\vec{k},\vec{k}')\frac{z}{2\pi}\right).$$

Here, z_0 denotes a plane in which the fundamental field $P_1\left(\vec{k}',z_0\right)$ is known. The observation depth is denoted by z, and the following substitution has been introduced:

$$\gamma(\vec{k},\vec{k}') = -K(\vec{k}) + K(\vec{k}') + K(\vec{k}-\vec{k}').$$

Where the K defined in (37) is used in [2, 27], and the Matlab implementation of *Propose* relies on $\mathcal{K}$, defined in (21).

Provided that the criteria for quasilinear propagation holds[5] for all combinations of x, y and z. Equation (40) can be used to compute the angular spectrum of P_2 for any plane z, in the half space defined by the transducer and its transmit direction. The only information needed is the known angular spectrum for the fundamental field in any one reference plane z_0.

5 p1 >> p2

Choice of reference plane z_0

The reference plane z_0 can in theory be arbitrary chosen, but the most efficient choice numerically is the focal plane z_d. The reason for this can be shown using the Fraunhofer approximation of the Huygens principle. This theory states that the FT of the fundamental field in the focal plane $P_1(x, y, z_d)$, is proportional to the spatial aperture function $A(x, y)$ [61, pp. 74-76].

By assuming symmetric aperture in the x and y directions[6], neglecting the proportionality factor and neglecting all terms that are constant in regard to x and y, the following relation of proportionality is developed in [2]:

$$P_1(k_x, k_y, d) \propto P(\omega)A(\frac{k_x dc}{\omega}, \frac{k_y dc}{\omega}) \otimes \mathcal{F}\left\{e^{\frac{i\omega}{2dc}(x^2+y^2)}\right\}. \quad (41)$$

The symbol $\otimes$ represents a two dimensional convolution in k-space, $P(\omega)$ is the temporal FT of the transmitted pulse and d is the focal depth of the transducer.

2.6 PROGRAMMING THEORY

In this section, different tools used in the fast *Propose* implementation are presented. The tools explained to a great level of details, in an effort to make the concepts easily accessable for readers who wish to try out the methods in their own projects.

First in this section a comparison formula is presented, which will be used to calculate relative time reductions between earlier presented versions of *Propose* [2, 1], and the additional optimizations implementations presented in this thesis. Then the concept of MEX files are explained and reasoned for, succeeded by Section 2.6.3, in which the Open Multi-Processing (OpenMP) [62] technology is thoroughly explained. The last subsection develops a mathematical foundation for conversion between single and triple indices used to access elements in at three dimensional *mxArray*, which is the C/C++ representation of a Matlab matrix.

2.6.1 *Formula for relative time reduction*

For relative comparisons undertaken in Chapter 5, between the different simulation times presented in Section 4.2. The follow-

6 $A(-x, -y) = A(x, y)$

ing formula has been used to calculate time reductions R expressed in percentage:

$$R = \frac{t_{high} - t_{low}}{t_{high}} \cdot 100\%. \tag{42}$$

2.6.2 *MEX files*

The MEX framework included in Matlab®[28] provides means to export data between Matlab and a compiled C/C++ or Fortran program. In C/C++ the framework consist of two header-files[7] which are included in the Matlab installation package. These header files contains information about Matlab data types and the entry function which is named *mexFunction* instead of the usual *main*.

A MEX function will operate like any other C or C++ program, with access to standard libraries or custom functions and templates. The code is compiled from the Matlab command window, and the compiled file is called from Matlab like any other Matlab function saved in the current workspace, using standard syntax.

For a complete guide on how to get started with MEX functions, the reader is referred to the online Matlab documentation [63].

Motivation for using MEX functions in a Matlab program

Matlab® is a powerful programming language in many ways. Especially because of powerful matrix and vector support. Advanced tools for visualization, profiling and debugging are also arguments for using the software. Because Matlab is running in a virtual machine framework, a program written in the Matlab language will be executable on any system with Matlab installed. This is both a strength and a drawback, because most of the program is interpreted at runtime. Built-in functions like matrix operations are pre-compiled and highly optimized and perform overall the same league as a C++ program (Appendix A). However, a user developed routine will require extensively more computational time than a compiled program [64]. Therefore such routines will have great potential for optimization by conversion to a compiled language.

7 The C/C++ header files are named *mex.h* and *matrix.h*.

Listing 1: C++ Simple OpenMP example.

```
#pragma omp parallell for <<optional clauses>>
for(int i=0;i<END;i++){
(...)
};
```

A for-loop construct is especially slow in an interpreted language, like Matlab, because each line inside the loop is interpreted at every iteration of the loop. In a compiled language like C or C++ this is not the case, because the code is translated to native machine code by the compiler only once, as the executable file is created. Later when the file is executed, no human programming language has to be interpreted, because the entire program consists of native processor and memory commands. [64]

A MEX function enables the Matlab programmer to take advantage of all the powers of Matlab, and still write computationally efficient programs.

2.6.3 *The OpenMP® compiler directives*

In a modern computer, the processors are able to execute many commands at the same instance, because the processor has multiple physical or virtual cores. For a program to take full advantage of this opportunity, it is important to tell the compiler which parts of the program that are independent of each other. Independent parts may be calculated simultaneously, utilizing the multi-core Central Processing Unit (CPU) in an efficient way.

OpenMP® [62] provides a powerful tool in C, C++ or Fortran programs, to inform the compiler about parallelizable parts of the code without altering the original structure. To parallelize an existing for-loop in a program, a "#pragma" statement (Listing 1) can be used to tell the compiler that the following loop may be executed in parallel. A main advantage in this approach is that any compiler without OpenMP support will ignore the statement completely and compile a serial program. This makes the source code backwards compatible, and one will not have to maintain multiple versions of the same source file. OpenMP is parallelizing with respect to logical CPU cores, meaning that it supports virtual threads [65, 66]. The remaining parts of this section is leaning exclusively on information obtainable in [67].

By default, OpenMP assumes all variables to be shared between the computational units (hereafter called threads). Sharing of the same variables is preferable for large arrays of independent data points, because it would consume a lot of memory to make individual copies of this data. Also because we are interested in having the different threads work together on the same array of data, instead of editing some random parts of individual copies. If however, there in a loop exists variables which are not supposed to be shared by the different threads, this can be specified by appending a *private(tempA, tempB, ...)* clause. If the programmer for clarity reasons wants to account for all variables in the loop, it is also optional to specify the *shared* variables in a similar manner.

The Schedule clause in OpenMP

A clause for OpenMP (Listing 1) that will be exploited in this thesis is the *schedule*. This clause might take one or two arguments, where the first one tells the compiler how the iterations in a loop are to be divided among the available threads. Argument number two is a number restricting the compiler to make every chunk of work contain a number of iterations equal to or greater than the given integer. If no second argument is given the default integer 1 is assumed, providing the compiler complete freedom to decide how many iterations every chunk will contain. In the present studies the default setting has been used as second argument in all implementations. The first argument, which defines the scheme after which the work is divided, can be either *static*, *dynamic*, *guided* or *runtime*.

A *static* scheme will divide the loop into equally sized chunks of iterations, each assigned to a specific thread. This scheme is specifically suited for computations where each loop requires approximately the same time to complete, and no thread needs to wait for others to finish. Such a case is shown in Appendix A.

The *dynamic* scheme divides the total number of iterations into equal chunks, but does not pre-assign any chunk to a specific thread. Under this scheme, every thread will start working on a random waiting tread whenever it finishes a chunk. This approach will introduce a bit more book-keeping and addressing of chunks and may therefore cause more overhead. It will be beneficial compared to the *static* work flow, if the iterations require variable amounts of time. An example where this scheme gives the best optimization, will appear in Section 4.2.

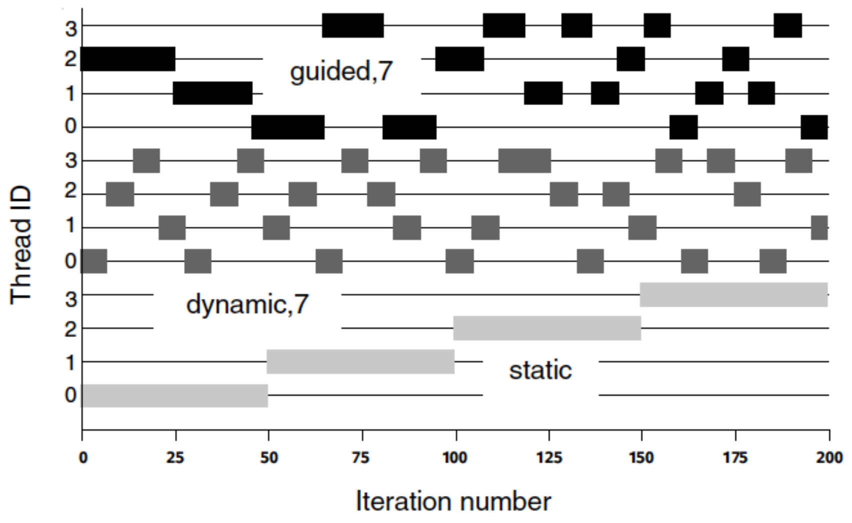

Figure 1: Graphical illustration of the different schedule settings in OpenMP, for a computer running through a loop of 200 iterations using four logical threads (0, 1, 2, 3). The thick lines distributed among the threads indicate iterations of the loop, assigned to the respective thread. Reprinted with permission from the MIT-Press, Cambridge, MA. [67, p. 88].

The *guided* scheme gives idling threads the same freedom to choose random waiting chunks of iterations, as is the case with dynamic scheme. Additionally, the guided scheme will have variable chunk size, starting with large number of iterations in each chunk and using small chunks in the end when the threads needs to synchronize their contributions to the work load and finish as simultaneous as possible. The *guided* scheme, like the dynamic, is best suited for computations with variable amounts of work in each iteration.

The *runtime* work scheme may be used if one want the schedule to be determined at runtime, using the *OMP_SCHEDULE* environment variable. This will make it possible to change schedule without re-compiling the code.

In Figure 1, the work flow is illustrated for the three possible work schemes. The *static* schedule will allocate chunks of data in a deterministic way while the others will allocate non-deterministic, depending on various factors, among which is the current load of the system.

2.6.4 *Parallelization of a triple nested for-loop*

The Matlab matrix is stored in an object containing a one-dimensional array and information about the size in various directions [68]. This provides an opportunity to access elements in a three dimensional matrix using a single index and one long for-loop instead of three indices and a nested triple for-loop. In my specialization project [1] a C++ function was developed calculating the zero-based single index $\mathcal{S}$ for a three dimensional matrix using three known subscripts i, j, k and the size I and J of in the first two dimensions in the matrix.

$$\mathcal{S} = i + j \cdot I + k \cdot I \cdot J. \tag{43}$$

In the preliminary study presented in Appendix B, it is shown that it will be computationally efficient to apply OpenMP on one extensive for-loop instead of parallelizing a smaller, outer loop in a nest of loops. This can be explained by noting that every iteration in the loop may require different amounts of work from the CPU thread. Merging nested for-loops will provide larger chunks of work, and thus even out the overhead. Another favorable effect of merging the for-loops is that a more extensive loop will easier lend itself to Graphical Processing Unit (GPU) parallelization at a later point. In GPU applications there will be a performance growth proportional to the number of iterations distributed among the computational kernels, favoring extensive for-loops above smaller ones [69].

Calculation of triple indices from the single index and sizes of a matrix

If the original triple subscripts are used in parts of the loop body, one has to calculate them from (43). This can be done by knowing two dimensions of the three dimensional matrix, in addition to the single index. Considering (43) we see that the single index $\mathcal{S}$ consists of three possible bulk sizes. First we have the Matlab column number i, secondly the number of rows expressed as $j \cdot I$ and at last the layer number in the third dimension, expressed as $k \cdot I \cdot J$.

If I or J is zero, the matrix would have no elements at all and would not exist. When I or J equals unity the matrix reduces to two dimensions. Having both I and J equal to unity reduces the array to a singleton matrix with k as the only running index. Assuming a three dimensional matrix[8], it is possible to find the

8 I.e. assuming that the lengths in all three dimensions I, J, K is greater than unity.

triple indices *i*, *j* and *k* in (43) from a known single index $\mathcal{S}$ and the two dimensional sizes by using congruence theory and back substitution [70].

Rearranging (43) and applying the modulo operator gives

$$\mathcal{R} = i + j \cdot I \equiv \mathcal{S} \mod I \cdot J, \tag{44}$$

where I and J still denotes the matrix size in the i^{th} and j^{th} dimensions, and $\mathcal{R}$ denotes the remaining elements in the matrix after all the full layers of the third dimension is subtracted. The symbol $\equiv$ is the congruence sign. From (44) the first index is found by taking $\mathcal{R}$ modulo the number of elements in a row

$$i \equiv \mathcal{R} \mod I. \tag{45}$$

The row-number is found by back substitution of i into a rearranged version of (44)

$$j = \frac{\mathcal{R} - i}{I}. \tag{46}$$

The third index is found through back substitution of i and j into a rearranged (43):

$$k = \frac{\mathcal{S} - i - j \cdot I}{I \cdot J}. \tag{47}$$

The C++ implementation of this algorithm is shown in Appendix C (Listing 3).

2.7 RELEVANT RESULTS FROM MY SPECIALIZATION PROJECT [1] AND SOME IMPLICATIONS

Some issues regarding the results presented in my specialization project [1] need to be presented here, because they have effected choices made about further optimization, conducted in the current study. The relevant issues are briefly presented in the following section.

An optimized version of the *Propose* Matlab program and the *sinc(x)* function (App. A) was made in my specialization project [1]. The optimization was done by implementing a C++ MEX version of the sub function governing nonlinear wave propagation in *Propose*. A MEX version of Matlab's built-in *sinc(x)* function was also created. The implementation was not optimized using OpenMP [67] and thus did not take advantage of multiple processor cores. It was found that a serial MEX implementation of *Propose* was running approximately two times faster than the Matlab implementation.

The calculation of sinc-values for all elements in a $100 \times 100 \times 100$ complex matrix was executed requiring 49% less computing time. Both observations were done using Computer 3 from Table 1 Section 3.1.

Further examination of the results showed that running Matlab under Windows 7 resulted in results contradictory to the original study. The MEX version of *sinc(x)* was found to perform slower computations than its corresponding Matlab version[9]. These findings suggests that the implementation made in [1] needs further optimization to become a general improvement.

Exploration of these issues would provide insights regarding the potential present in OpenMP, for optimizations in the MEX version of *Propose*. Therefore a preliminary study has been undertaken, to enlighten issues regarding *sinc(x)* execution time on different OSes, with and without MEX and OpenMP. The results are presented in Appendix A, and led to insights used for the OpenMP optimizations of *Propose* (Section 3.3).

2.8 THE MECHANICAL INDEX (MI)

If ultrasound is transmitted at high power though biological tissue, there is a risk of inducing cavitation [71]. Cavitation is a process where the local minimum pressure is lower than the current boiling pressure of the propagation medium[10], causing bubbles to form in the fluid. As the pressure rises again, the bubbles will implode fast enough to satisfy the criteria for an adiabatic process. This quick implosion may locally give rise to temperatures up to thousands of kelvin [72].

The MI is developed to give a qualitative indication of the likelihood that a certain ultrasound pulse may induce cavitation, and is used to define threshold values under which no danger is imposed [73]. This index is calculated using the following formula [73]:

$$\mathrm{MI} = C_{\mathrm{MI}} \frac{P_r}{\sqrt{f_c}}, \tag{48}$$

where f_c is the pulse center frequency in MHz and P_r is the peak rarefaction pressure in MPa, often found at focal depth

9 Thanks to Alfonso Rodriguez Molares at the department of Circulation and Medical Imaging, for making this observation.

10 The boiling temperature of a fluid depends proportionally on its pressure, at lowered pressure, the boiling temperature will be lowered as well

or shortly ahead of it. To make the index dimension less, the factor C_{MI} of magnitude $1 MHz/MPa$ is included in the formula. When the measurements or simulations are performed in homogeneous tissue like water, it is derated by a factor of $0.3 dB/cm/MHz$. For a center frequency at 1.7 MHz and focal depth at 7 cm, this deration factor will be 1.497.

As the pressure amplitudes in a pulse increases, the nonlinear effects also will rise. The MI serves as a useful mean to express which amplitude is used in a simulation and thus also how much nonlinear effects that are present. The quasilinear approximation used in *Propose*, will give a less accurate estimates as the transmit amplitude rises (Section 2.5.2). Therefore it has earlier been found less correlation between *Abersim* and *Propose* simulations, as the MI was increased [39]. Some more details about this article is given in Section 2.1.2.

3

METHOD

3.0.1 *Explanation of coordinates referred to in plots and comments*

The coordinates referred to in this study is related to each other in the following way:

- All z values in the plots have been rendered from the transducer ($z = 0$) with positive z in the direction of propagation.
- The azimuth plane corresponds to the Cartesian xz-plane where all y values are zero.
- The elevation plane are equivalent to the Cartesian yz-plane, where all x values are zero.

When retarded time is used, this indicates a time laps observed from stationary plane with constant z value.

3.1 COMPUTERS AND CONFIGURATIONS

The various computers used in this thesis are listed in Table 1. The three computers represent three different price levels, and are comparable in the sense that they all are relatively standard consumer market computers.

Computer 3 in Table 1, has also been used for simulations and performance tests in my specialization project [1], and was chosen to provide an extra verification of the results obtained there. The use of this computer also provided a good mean to compare the optimization achieved to results in the specialization project. Computer 2 is a standard Linux® desktop available at the Department of Physics as part of a desktop cluster. Computer 1 is a high-end Linux desktop, used at the Department of Circulation and Medical Imaging for simulation of computationally demanding problems. Computer 1 and 2 computers were chosen as good middle-end and high-end alternatives running another OS than Computer 3, thus not restricting the generality of the results to a specific system. Because they also have different Linux versions, namely Ubuntu 12.04 and Fedora 18, they were expected to provide extra insight to variations between different Linux based systems. No Windows® computer were chosen, because the available Windows version of Matlab[28] did not support any open-source compilers with OpenMP awareness.

	High-end desktop PC	Standard desktop PC	MacBook® Pro
Computer number	1	2	3
OS	Ubuntu® 12.04	Fedora™ 18	OS X® 10.9.1
CPU type	Intel® Xeon®	Intel® Core™ i7	Intel® Core™ i5
CPU architecture	x86_64 bit	x86_64 bit	x86_64 bit
CPU cores (physical)	12	4	2
CPU cores (logical)	12	8	4
CPU clock frequency	2.67 GHz	1.60 GHz	2.30 GHz
Total RAM	96.7 Gb	12.0 Gb	8.0 Gb
Matlab version	2011b [74]	2011b [74]	2013b [28]

Table 1: Overview of computers and specifications. Machine numbering in this table will be used for later reference.

3.1.1 *Simulated systems*

In the comparison studies (Section 3.2) the simulated system has been set in conjunction with [26] to make better comparisons to this work. The transmit frequency only, has been chosen differently (2.7 Mhz), because this study is not restricted by limitations in experimental measurements, and the chosen transmit frequency correspond better to a typical cardiac probe. Details of the setup used for comparisons is presented in Table 2.

The system in used in the performance studies (Section 3.4) is chosen in conjunction with my specialization project [1] to make the optimization results directly comparable to those obtained there. Table 3 display the details in this setup.

3.2 COMPARISON OF *propose* AND *abersim*

To qualitatively compare *Abersim* and *Propose*, the system in Table 2 has been simulated using both methods. Comparison of normalized RMS beam profiles and sound fields have been performed at three different simulation depth in both azimuth and elevation directions. The near field initial signals are also compared.

In *Abersim*, the propagation medium was set to *water* at 293 K and *Propose* was set to simulate homogeneous medium with no attenuation. Sampling frequencies have been set to 40 MHz and 23 MHz for *Abersim* and *Propose* respectively.

The *Abersim* plots were shifted backwards in time to align better with the *Propose* version. This time difference between the plots depends on the range in the z-direction chosen in *Propose*, and is of no special significance.

To display agreement between the new *Propose* implementation and *Abersim* two types of figures have been made. The RMS beam profiles were chosen because they provide a precise comparison between the magnitude levels in the two simulated plots, because they are normalized. Any overestimation in the center lobe will be apparent as lowered side lobes, because they are normalized by a center lobe which is too high. Decibel scale have been chosen on the magnitude scale, to better visualize weak side lobes together with the center lobe.

The grayscale field plots used in Section 4.1.4 were created by normalizing the sound field matrix, and taking a logarithm of all absolute values, with dynamic range, 40 dB. To visualize the phase of the waves, and not just the absolute values, each element of the matrix was afterwards multiplied by its original sign. Once again, decibel scale is used to enhance the weak side lobes, in the same plot as the center lobe.

3.2.1 *Filtering*

Since the *Abersim* signal is generated in the time domain, it contains all frequency components. To be able to compare the fundamental and second harmonic frequency individually, bandpass filters have been applied. The fundamental frequency was extracted using a Butterworth filter [75] of 6th order with center frequency $f_c = 1.7$ MHz and a total bandwidth of $f_b = 1.3$ MHz. The second harmonic frequency band was extracted using a Butterworth filter of 8th order with center frequency $f_c = 3.4$

Setting	Value or information
Aperture shape	Rectangular
Transducer focal depth, azimuth	70 mm
Transducer focal depth, elevation	70 mm
Transmit aperture, azimuth	22 mm
Transmit aperture, elevation	13 mm
Transmit apodization	none
Transmit center frequency	1.7 MHz
Bandwidth of transmitted pulse	1.1 MHz
Dynamic range in plots	40 dB
Attenuation	none

Table 2: Compared system, simulated both in *Propose* and *Abersim*.

MHz and bandwidth $f_b = 1.0$ MHz. The magnitude response of the two filters are displayed in Figure 2, together with the raw-signal simulated in *Abersim* at depth $z = 2.5$ mm.

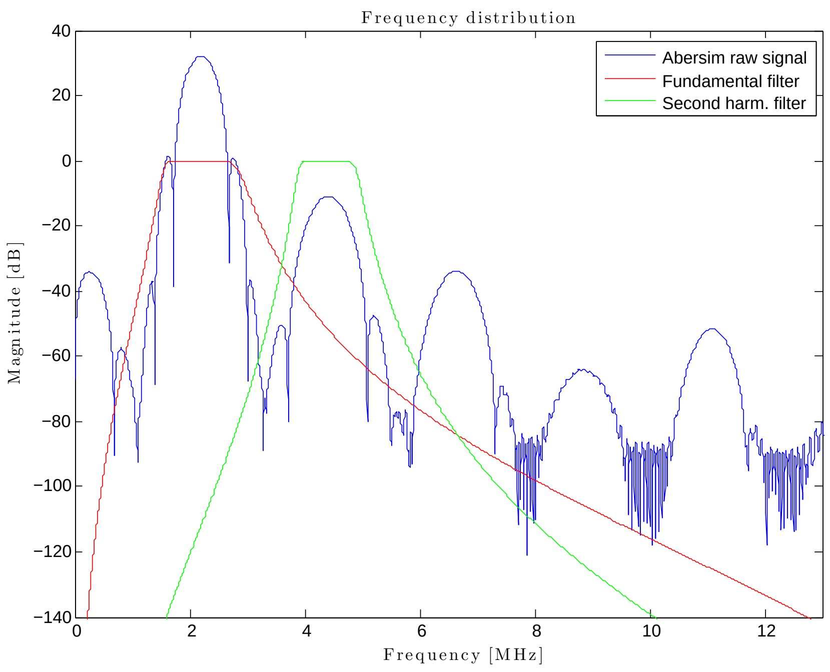

Figure 2: Magnitude respond for the filters applied to the signal simulated in *Abersim*. The simulated raw-signal at depth $z = 2.5$ mm is also included to illustrate the adequacy of the chosen filter characteristics.

3.3 IMPLEMENTATIONS

3.3.1 *New implementations in the* Propose *software*

An optimized implementation of the nonlinear wave propagation routine in *Propose* has been developed based on the original Matlab implementation [2] and on the MEX implementation created as part of my specialization project [1].

The choice to address the subroutine *prop2harm*, which is created for calculating the second harmonic field, is justified thoroughly in [1] as a result of initial profiling that identified this subroutine as the most computationally demanding part of the program. Much of the computational time in this function is spent in a triple nested for-loop[1] which is known to be a bottle neck in Matlab programs (Section 2.6.2).

As a basis for the new, fast implementation of nonlinear wave simulation, the MEX version *prop2harm*, which initialy was presented in my specialization project [1]. To obtain a generally high performance, which did not depend on OS or OpenMP (Section 2.6.3) had been qualified as a useful tool through the preliminary study presented in Appendix A.

Some structural optimizations have also been undertaken in the code. A complete copy of the source code, accompanied with an overview of updated features[2] is given in Section C.3. The optimizations made in the current study have been explored using three different computers and operational systems (Table 1) and the results are presented in Section 4.2, Figure 14.

Another structural optimization in the C++ code was found through a preliminary study presented in Appendix B. That study was undertaken to compare two different ways to access elements in a 3-dimensional matrix. This study lead to an implementation choice where a triple for-loop in *prop2harm* (Listing 5) was replaced by a single, more extensive one. Because some of the triple indices also are needed inside the loop, a function shown in Listing 3 (Appendix C) was made to extract them from the single index used in the new loop. Some theoretical insights regarding this, has been presented in Section 2.6.4.

In the implementation phase it became evident that the new, single for-loop slowed down simulations utilizing symmetrical

1 Calculating a triple integral in Fourier space (Eq. 40).

2 *prop2harm* v. 2.0

Setting	Value or information
Aperture shape	Superelliptic (elliptic exp=4)
Transducer focal depth, azimuth	90 mm
Transducer focal depth, elevation	86 mm
Transmit aperture diameter, azimuth	22 mm
Transmit aperture diameter, elevation	13 mm
Transmit apodization, azim. and el.	0.1 and 0.1 (R value in Tukey window)
Transmit center frequency	1.67 MHz
Bandwidth of transmitted pulse	1.0 MHz
Resolution, dx - dy - dz	0.150 mm 0.150 mm 0.100 mm
Phase correction (non-spherical surface)	ON

Table 3: Simulation setup used for the performance measurements (Section 4.2). This table and setup has previously also been used in my specialization project [1].

properties[3]. The reason for this was that the symmetrical properties is built into the runtime determination of the triple-loop indices, meaning that only 1/4 of the data points are accessed and calculated inside the loop[4]. A single for-loop is accessing the data in memory ordered sequence [68] and do not have the option to adjust limits of sub-loops at runtime. This forces the processor to access every data point in the matrix, and start by checking whether it is to be calculated, or can be disregarded. The resulting overhead arises as 3/4 of all iterations subscribed to a certain thread is aborted. To maintain the possibility for a user to specify that symmetry properties apply, version 2.0 of *prop2harm* includes an if-clause choosing a triple loop if symmetry utilization is requested, and the single loop otherwise.

For complex computations in the new MEX implementation, the standard template library *complex.h* for C++ has been used. The C++ code was compiled using the GCC 4.8.1 for Linux and the GCC 4.9.1 for OS X. On OS X the compiled file requires GCC 4.7 or a newer version to be installed on the computer, otherwise it will not execute. The version compiled for Linux is more portable between computers, the same MEX file is working on both Fedora and Ubuntu.

3 The slowdown was of magnitude 3-4 times compared to the triple-loop calculation time

4 The rest are copied afterwords

3.4 SIMULATION PERFORMANCE TESTS

To determine the effect of the new optimizations, the computational time of a second harmonic field containing $66 \times 38 \times 11$ matrix elements were measured using Matlab's *tic-toc* function. Each simulation were executed 2500 times, and the uncertainty was calculated using the sample standard deviation on the resulting set of data [76]. The results are presented in Section 4.2. All three computers (Table 1) were set to simulate the system summarized in Table 3. The reason for performing the same tests on various computers was to explore the effect of MEX optimization on different OSes, ensuring that the results were not limited to a specific one.

For reference, both the original Matlab implementation [2] and a serial MEX implementation with no OpenMP was measured. The serial implementation was not the version used in my specialization project [1], but the version included in this thesis (Section C.3) with all OpenMP calls commented out at compile time. The new implementation was measured for all three OpenMP schedules (Section 2.6.3). This was done to give a picture of the variations among them, as well as determine which one is best suited for the problem at hand.

4 RESULTS

In the first section of this chapter, simulation outputs from *Propose* and *Abersim* are presented in comparable figures. Results presented here serve as foundation for the discussion in Section 5.1, regarding the accuracy of simulated sound fields, compared to the state-of-the-art simulation tool, *Abersim*. Section 4.2 is dedicated to the performance results obtained for the new, fast implementation of nonlinear wave simulation, and timing of older versions of the program, for comparison.

The three depths at which figures have been produced is $z = 2.5$ mm, $z = 70$ mm and $z = 90$ mm, where the focal depth in all cases has been at $z = 70$ mm.

4.1 COMPARISON OF PROPOSE AND ABERSIM

Figures containing comparable simulations from *Abersim* and *Propose* will in the following be presented for three different observation depths. Both azimuth ($y = 0$) and elevation ($x = 0$) sections have been included in the figures.

Note that *Abersim* applies a window function to the fundamental signal at every propagation step, to suppress peripheral side lobes [44]. This will especially be visible in the RMS beam profiles.

All the comparison plots are simulated using the set up in Table 2, Section 3.2.

4.1.1 *Determination of the Mechanical Index (MI) in Abersim*

The MI for all presented figures, are calculated from the peak rarefaction P_r at focal depth, which is shown in Figure 3. Using the formula (48) and deration factor given in Section 2.8, the MI is found to be 0.13. In *Propose*, the MI is not accounted for, but it will affect the output in *Abersim* simulations, therefore it is included in the figure labels.

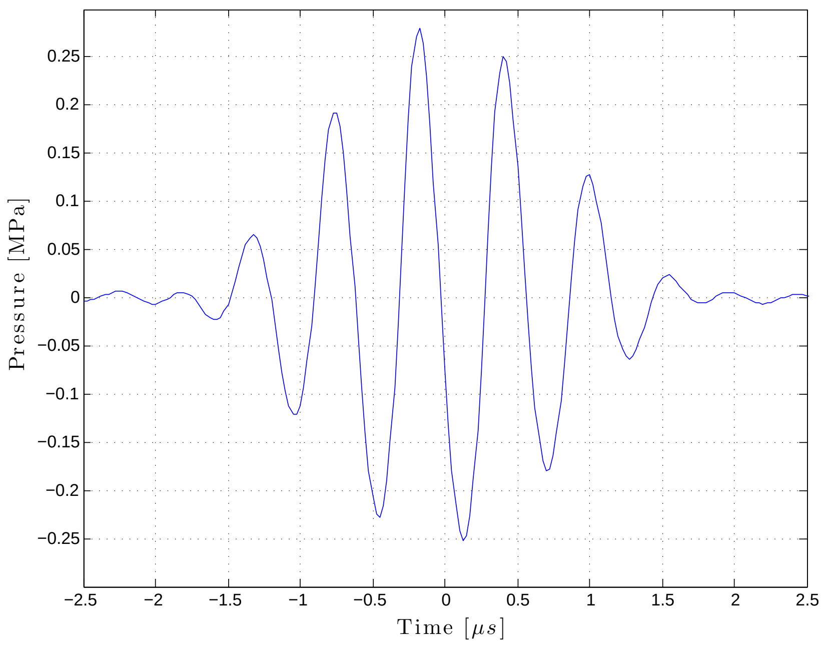

Figure 3: Nonlinear *Abersim* pulse at focal depth, $z = 70$ cm. Transmit frequency is 1.7 MHz, propagation medium is water, and peak rarefaction value is 0.25 MPa. This leads to a MI of 0.13, which has been kept constant for all the comparisons in this chapter.

4.1.2 *Comparison of fundamental starting signal*

Initially the fundamental signals have been compared in Figure 4 to verify that both simulations start with signals of comparable frequency and amplitude. The figure shows a good agreement in frequency and pulse length. There are minor differences in the amplitude at the start and the end of the pulse. The *Abersim* pulse is of highest amplitude at the start of the pulse and the *Propose* pulse is highest at the end.

4.1.3 *Normalized RMS beam profiles*

In this section, normalized RMS beam profiles in the fundamental and second harmonic frequency bands, are presented for all three comparison depths. In all beam profiles, the *Propose* simulation is displayed in red, and the blue line showed the corre-

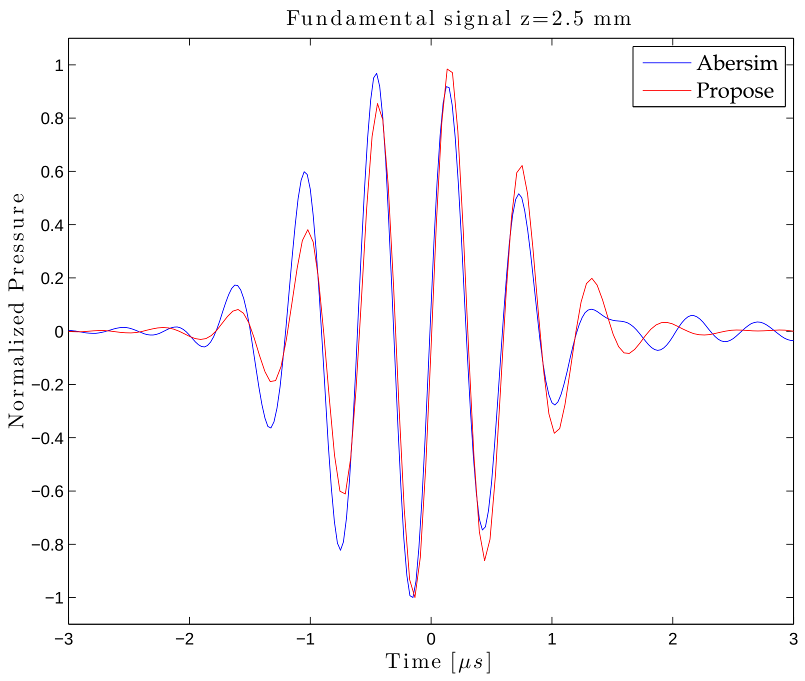

Figure 4: Simulated fundamental signals in water, at depth $z = 2.5$ mm, and MI $= 0.13$. Red and blue line represent *Propose* and *Abersim* respectively.

sponding *Abersim* simulation. The vertical scale is logarithmic, ranging from 0 dB to −40 dB for the fundamental profile, and from 0 dB to −70 dB for the second harmonic frequency band.

The normalized RMS beam profiles at depth $z = 2.5$ mm ared presented in Figure 5. In this figure the sides of the central lobe display some similarities for intensities above −25 dB. In the fundamental profiles, and above −30 dB in the second harmonic case. Some agreement is present but limited in the sense that *Abersim* include details in the beam profile which is are missed by *Propose*. On the top of the profiles the central intensities are overestimated in *Propose* by an average of 1.8 dB for the fundamental intensities and 2.6 dB in the second harmonic case. Along the sides, the profiles sometimes coincide, but in the fundamental azimuth plot there is approximately 5 dB disagreement, along the right slope of the beam profile.

At focal depth (Figure 6), the normalized RMS center lobes from *Propose* and *Abersim* coincide for both azimuth and elevation in both frequency modes. The first side lobes also coincides quite well. From the figure there is found less than 1.5 dB disagreement for all fundamental side lobes above −20 dB, and for all second harmonic side lobes above −35 dB. For lower intensities, the side lobes are generally underestimated by more than 1.5 dB in version *Propose*.

Normalized RMS beam profiles obtained at depth $z = 90$ mm are presented in Figure 7. The second harmonic azimuth plot in this figure display no disagreement larger than 2 dB for side lobe intensities above −40 dB, and no disagreement larger than 1 dB for side lobes above −35 dB. The second harmonic elevation plot disagree no more than 2 dB for side lobes above −32 dB, and with no more than 1.5 dB for side lobes above −26 dB. For the fundamental plots, the any disagreement is not higher than 1.5 dB, for side lobes above −23 dB.

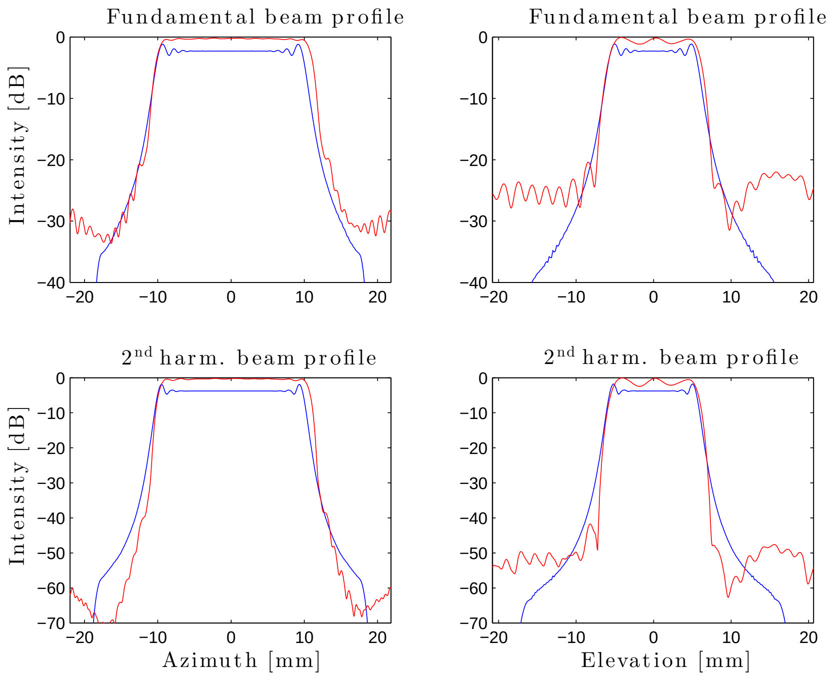

Figure 5: Normalized RMS beam profiles displayed for azimuth and elevation at depth $z = 2.5$ mm and MI $= 0.13$. Red and blue line represent the beam profile from *Propose* and *Abersim* respectively. Note the different decibel scales.

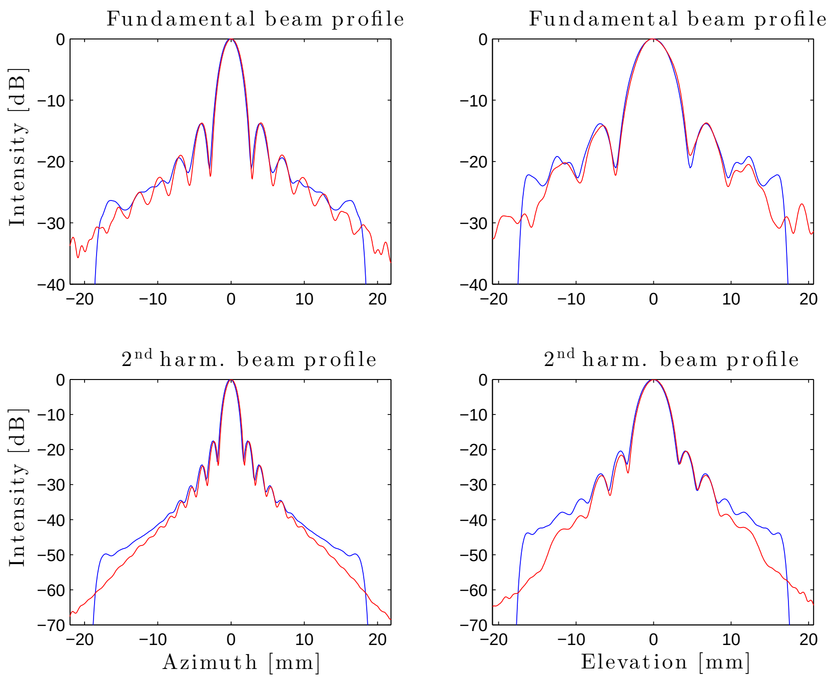

Figure 6: Normalized RMS beam profiles displayed for azimuth and elevation at focal depth $z = 70$ mm and MI $= 0.13$. Red and blue line represent the beam profile from *Propose* and *Abersim* respectively. Note the different decibel scales.

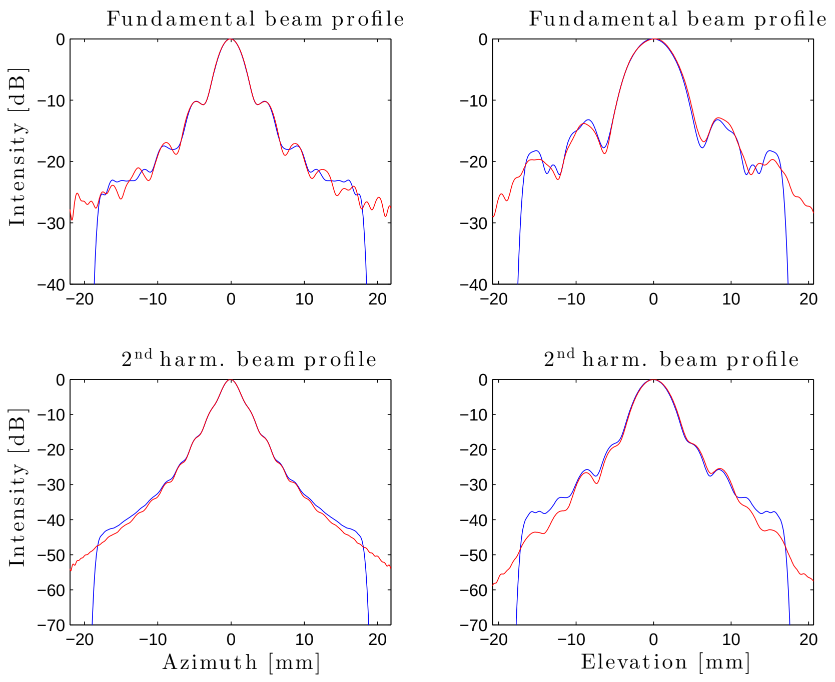

Figure 7: Normalized RMS beam profiles displayed for azimuth and elevation at depth $z = 90$ mm and MI $= 0.13$. Red and blue line represent the beam profile from *Propose* and *Abersim* respectively. Note the different decibel scales.

4.1.4 *Field comparisons*

Comparable sound fields presented in Figures 8–13 are illustrated using logarithmic grayscale plots. The magnitude of the presented sound fields are normalized to one, and are given a logarithmic dynamic range of 40 dB.

The sub figures are ordered in the same manner for all Figures 8–13. *Abersim* simulations are displayed in the upper row and *Propose* simulations are shown in the lower row. Cross sections in azimuth is shown in the left column, and elevation is located in the right column.

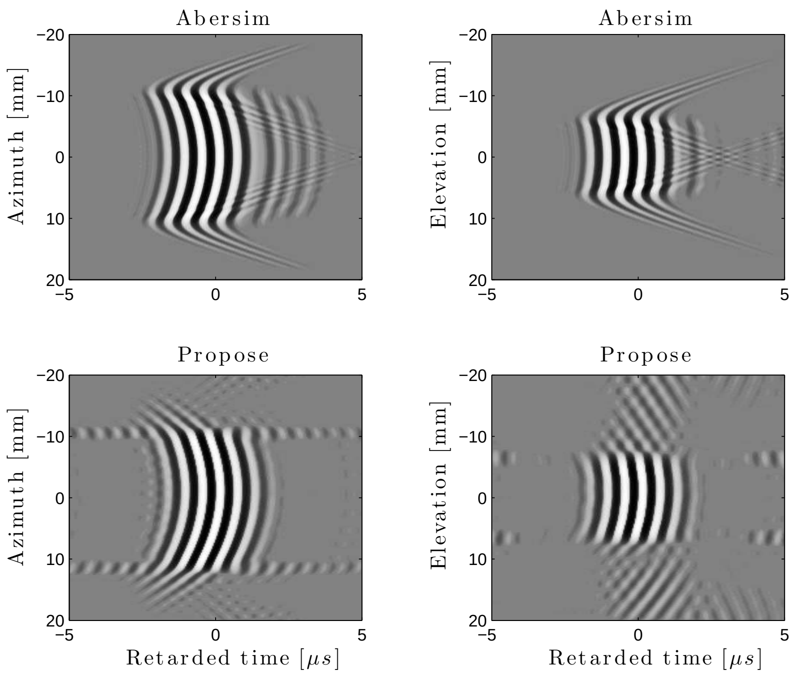

Figure 8: Field plot of the fundamental (1.7 MHz) pulse in retarded time, at depth $z = 2.5$ mm and MI $= 0.13$. The magnitudes are normalized to one, and the logarithmic grayscale is given a dynamic range of 40 dB. The figures display simulations in azimuth (left) and elevation (right) directions, simulated using *Abersim* (top) and *Propose* (bottom).

In Figure 8 the field at $z = 2.5$ mm is displayed for the fundamental frequency band. Both pulse shape, length and frequency are comparable. The edges surrounding the pulses are not similar, this is is due to the different nature of the methods. The interference patterns surrounding the fundamental field simulated in *Propose* are caused by the periodicity of the FT. At

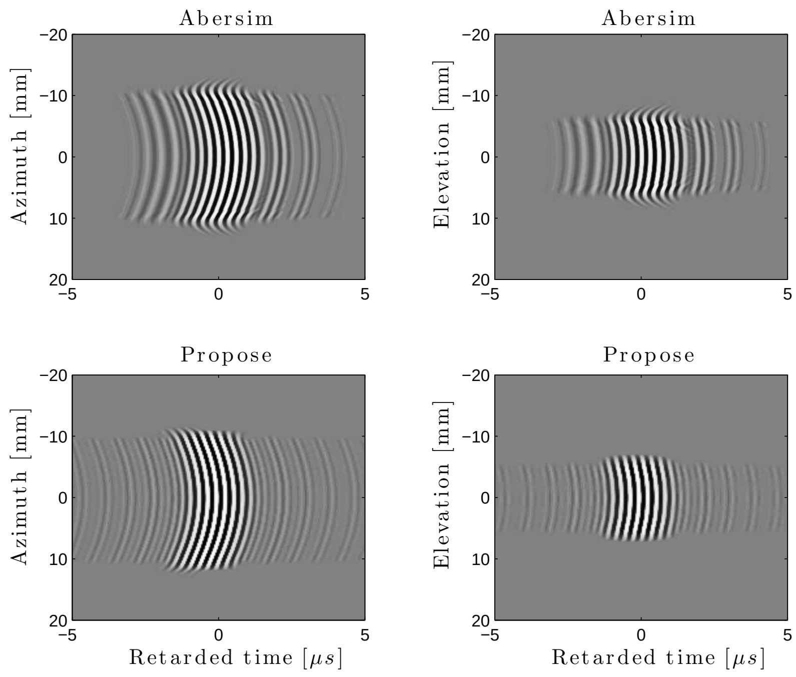

Figure 9: Field plot of the second harmonic (3.4 MHz) pulse in retarded time, at depth $z = 2.5$ mm and MI $= 0.13$. The magnitudes are normalized to one, and the logarithmic grayscale is given a dynamic range of 40 dB. The figures display simulations in azimuth (left) and elevation (right) directions, simulated using *Abersim* (top) and *Propose* (bottom).

the same depth, simulations of the second harmonic field is displayed in Figure 9. In this plot the edges are more similar, though the *Propose* plots contain less details about the edge shapes, than the corresponding *Abersim* simulations.

The fundamental field at focal depth ($z = 70$ mm) is displayed in Figure 10. Pulses agree well in regard to both shape and frequency. Traces of the slightly lowered side lobes for *Propose* can be observed at the edges of the sub figures. Interference patterns can be seen in both propose plots, these are traces of the neighboring solutions shaped by the IFT. Figure 11 show the second harmonic field at focal depth. Here, the shape, and phase pulses are also in agreement, but the underestimated side lobes has become more apparent. This too, is in agreement with the earlier presented beam profiles at focal depth (Figure 6). They predict a more evident disagreement in the beam profiles, for the second harmonic, than in the fundamental frequency band.

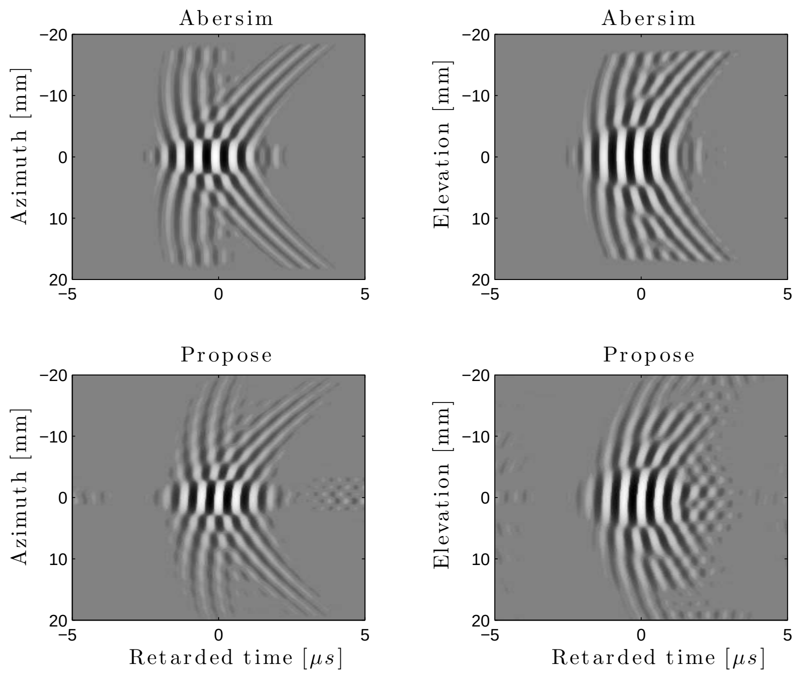

Figure 10: Field plot of the fundamental (1.7 MHz) pulse in retarded time, at focal depth $z = 70$ mm and MI $= 0.13$. The magnitudes are normalized to one, and the logarithmic grayscale is given a dynamic range of 40 dB. The figures display simulations in azimuth (left) and elevation (right) directions, simulated using *Abersim* (top) and *Propose* (bottom).

Figure 12 display the fundamental field at $z = 90$ mm. Again, the pulse shape, length and amplitude are in good agreement. Other than some traces of the IFT visible in in the propose plots. In accordance with the beam profiles for $z = 90$ mm (Figure 7), there are little traces of underestimated side lobes here. Also for the two azimuth plots one can see that even details like height levels, following the side lobes, are accurately represented in the new *Propose* method. The second harmonic field at $z = 90$ mm is presented in Figure 13. Here to pulse shape and phase are in good agreement. The *Propose* side lobe levels are underestimated more for the elevation case, than the azimuth direction. This, again is in accordance with effects shown in the beam profile for $z = 90$ mm (Figure 7).

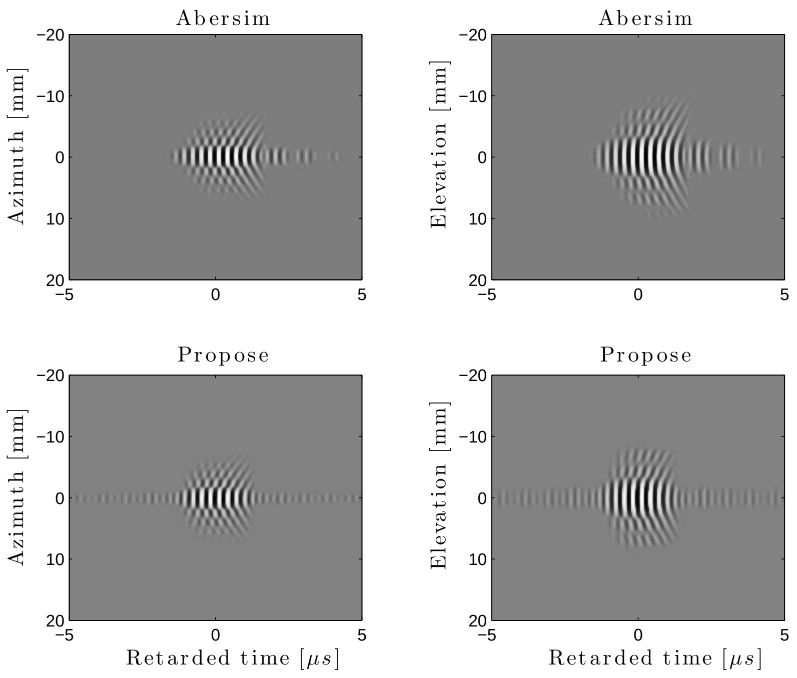

Figure 11: Field plot of the second harmonic (3.4 MHz) pulse in retarded time, at focal depth $z = 70$ mm and MI $= 0.13$. The magnitudes are normalized to one, and the logarithmic grayscale is given a dynamic range of 40 dB. The figures display simulations in azimuth (left) and elevation (right) directions, simulated using *Abersim* (top) and *Propose* (bottom).

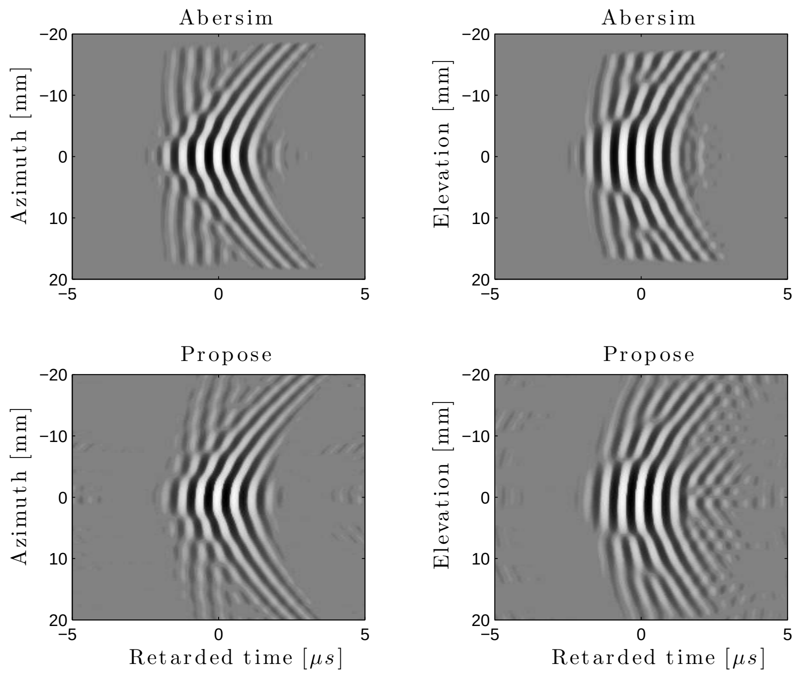

Figure 12: Field plot of the fundamental (1.7 MHz) pulse in retarded time, at depth $z = 90$ mm and MI $= 0.13$. The magnitudes are normalized to one, and the logarithmic grayscale is given a dynamic range of 40 dB. The figures display simulations in azimuth (left) and elevation (right) directions, simulated using *Abersim* (top) and *Propose* (bottom).

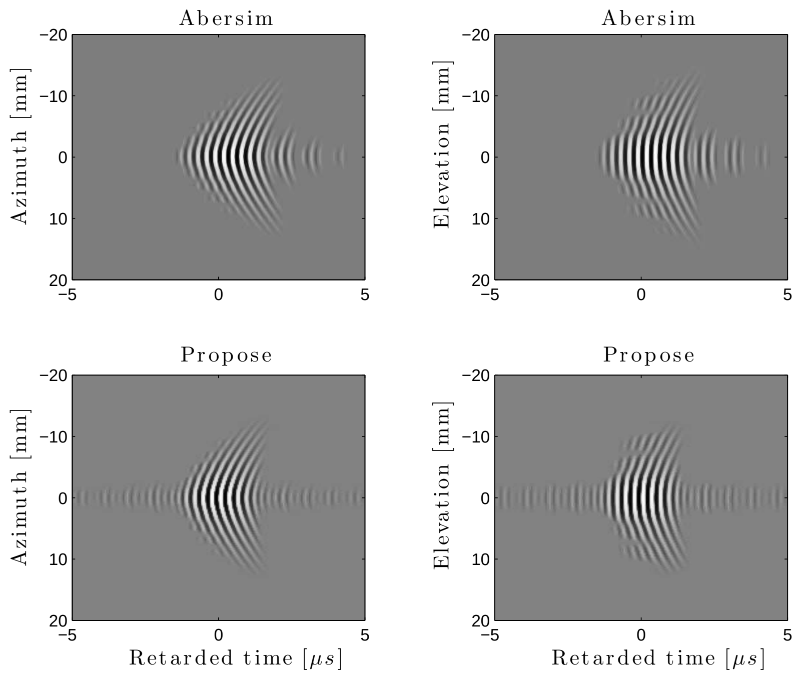

Figure 13: Field plot of the second harmonic (3.4 MHz) pulse in retarded time, at depth $z = 90$ mm and MI $= 0.13$. The magnitudes are normalized to one, and the logarithmic grayscale is given a dynamic range of 40 dB. The figures display simulations in azimuth (left) and elevation (right) directions, simulated using *Abersim* (top) and *Propose* (bottom).

4.2 PERFORMANCE OF THE FAST IMPLEMENTATION

In this section, the results of the optimization studies are presented. The new implementation is based on a Matlab implementation of the *Propose* presented in [2], and on initial optimizations conducted as part of my preliminary specialization project [1]. Method and implementation details are presented in Section 3.2, where the computers and the simulated system are specified in Table 1 and 3 respectively. The complete, updated MEX source code are displayed in Appendix C. All time recordings will be presented in seconds, and sample standard deviations have been included in Figure 14 as error bars.

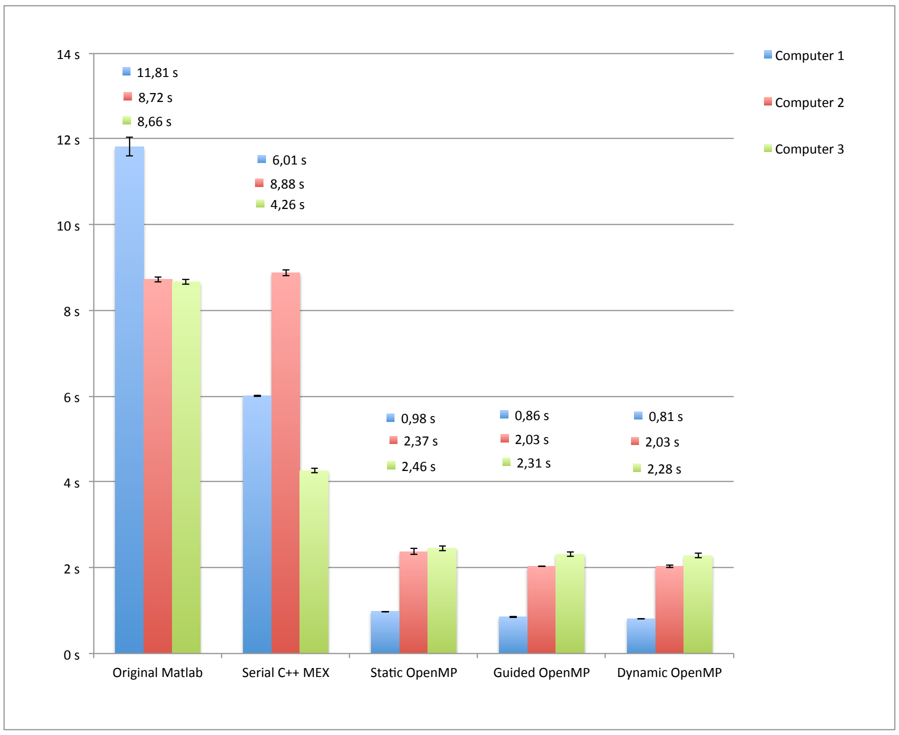

Figure 14: Computation time required for simulation of an ultrasound field in the 2nd harmonic frequency band containing 27588 matrix elements. The numbers used to denote different computers correspond to Table 1. The first group of bars display the performance of the initial *Propose* implementation presented in [2].

The results displayed in Figure 14 shows simulation time for the reference system specified in Table 3. The colors are assigned to each of the three computers which has been used for the benchmarking (Table 1).

The original Matlab implementation of *Propose* [2] represent an initial situation, to which the other implementations may be compared. The initial computational time requirements (8.66 s, 8.72 s and 11,81 s) are displayed in the leftmost group of bars (Figure 14).

The second group of bars in Figure 14 shows the performance of the new implementation, when OpenMP is disabled. Since the new implementation is based on the MEX file presented in my specialization project [1], this serial measurement is expected to be a good representation of the performance provided in that first stage of optimization. More generally, it illustrates the difference between enabling and disabling OpenMP parallel-processing for various OSes and computers, using programs which are otherwise identical. The fastest computer here performs the computations in 4.26 s, while the slowest computer is finished after 6.01 s.

The three rightmost groups of bars in Figure 14, originate from the new implementation, presented in this thesis. The three OpenMP schedules display quite similar results, and the simulation time required, using any OpenMP setting, range from 0.81 s to 2.46 s for the three computers.

The simulation at depth $z = 90$ mm, discussed in Section 4.1 was calculated in *Abersim* within 31560 s, in the new implementation of *Propose*, the plane was simulated in 25 s. The two programs are not directly comparable in respect to time consumption, but the numbers presented here can be used for approximation purposes.

5

DISCUSSION

This chapter will first focus on the agreement between the new, fast implementation of *Propose* and the state-of-the-art simulation tool, *Abersim*. A discussion of improvement in performance for the new method, compared to earlier versions are undertaken in Section 5.2, evaluating whether the aim of making a fast implementation has been reached. In the end of the chapter a short discussion on the performance compared to *Abersim* is given, followed by a section on limitations to the quasilinear method, and a note on a possible error.

5.1 COMPARISON OF PROPOSE AND ABERSIM SIMULATIONS

This section starts with a discussion regarding RMS beam profiles comparison plots (Figures 5-7), then the field plots (Figures 8–13) will be discussed.

The comparison results presented in Section 4.1 has provided detailed figures describing the agreement and disagreement between *Propose* and *Abersim*.

At focal depth $z = 70$ mm and at depth $z = 90$ mm the beam profiles presented in Section 4.1.3 showed good agreement in the simulations performed with *Propose* and *Abersim*. The beam plots agreed in center lobes, and also in the side lobes down to at least -20 dB and -35 dB. The underestimation in the lowest side lobes is expectable, and can be explained by the nature quasilinear approximation [25] used in *Propose* (Section 2.5.2). The quasilinear approximation neglects losses to higher order frequency modes, and thus perform an over estimation of the center lobe. In a normalized plot this will appear as an underestimation of the low side lobes. For the applied MI of 0.3, the nonlinear effects in the beam profiles in Figures 6 and 7 show good comparison with the results presented in [39]. In that study a comparison between the quasilinear method was compared to an early version of *Abersim* [40] for various MIs. Other articles have also confirmed that some over estimation is expectable, but that it can be tolerated for propagation in media where the nonlinear effects are small [2, 26, 27].

In the beam profiles at depth $z = 2.5\mathrm{mm}$ (Figure 5), the *Propose* simulations include considerable noise levels outside the main beam profile. Non of the beam profiles fits well with the *Abersim* version. This noise appears because we are close to the rectangular transducer, meaning that the FT of the sound field is close to a the FT of a rectangle, which is an infinite *sinc* function in the fourier domain [61, pp. 74-76]. Because *Propose* has a finite range in the frequency domain, and hence will cut the *sinc* function somewhere, some of the rectangular shape will be lost after the IFT back to space and time. Thus, obscuring the simulation. For *Abersim*, this issue is avoided, because the field is described directly in real space and time. This will be noted as a limitation to the method, but it might not be an important one, because this close to the transducer, more accurate simulation method would also produce results within a reasonable amount of time.

As pointed out in Section 4.1.4, the effect of overestimation in the *Propose* center lobe appears to be stronger in the second harmonic case, than for the fundamental mode. A center lobe overestimation similar to the one present for the fundamental also expected and found for the second harmonic case. The second harmonic frequency mode will also be lose energy to higher order frequency modes, as the effects of nonlinearity is increased. This loss is not accounted for in the *Propose* method, therefore an overestimation will occur. According to [26], there is also an even more important contribution to the overestimation in the second harmonic case. The second harmonic frequency band, will be overestimated in *Propose* because it is calculated from an already overestimated fundamental field. This might also argue for more visible effects in the normalized plots, for the nonlinear case. The apparent stronger effects might also be caused by the relative numbers in view, because the side lobes are already small, they are therefor easily affected by an overestimated center lobe. There are figures supporting the hypothesis in [27, 39].

Section 4.1.4 present six field comparison figures which all provide good agreement between *Propose* and *Abersim*. Even the simulations at depth 2.5 mm, which earlier was disregarded in the discussion, has provided good agreement in the field comparison plots (Figure 8 and 9). All the other field plots, also show good comparison, and at those instances where they disagree the side lobe amplitudes, this is in agreement with the beam profiles in Section 4.1.3.

In the *Propose* simulations, an interference pattern is visible outside the main pulse. As mentioned in Chapter 4, this is because the simulations are performed in the Fourier domain, and will be reproduced in a periodic pattern. Neighbor pulses will then be able to interfere with each other, giving rise to patterns like the one clearly visible in the elevation column of Figure 10. As *Abersim* is applying a window function between each propagation step, the neighboring images are thus suppressed and will not provide the same amount of interference. The phenomena is more visible in the fundamental field when the absolute amplitudes are high and the side lobes reach far out from the center lobe. This is when the side lobes of neighboring images are strong enough to create constructive interference.

The plots show all in all good agreement, given the limitations expected for the quasilinear approximation. Thus, *Propose* version compared to *Abersim* can be said to produce reasonable results.

5.2 COMPUTATIONAL PERFORMANCE OF OPTIMIZED IMPLEMENTATIONS

In this section, the results presented in Section 4.2 will be thoroughly discussed and examined. First, the results obtained for different OpenMP implementations are discussed. Some relative improvements and differences are pointed out by using (42) to compare different simulation times recorded in Figure 14. Then general observations and discussions are presented, succeeded by a subsection regarding the required simulation time in *Propose* compared to the one needed in the *Abersim* implementation.

Figure 14 display the OpenMP simulation times for each of the three possible schedule settings presented in Section 2.6.3. The most important result from these three groups, is that there are very little differences between the various schedule settings. Absolute differences between the different schedules are small, but the relative difference between the best and the worst choice is 21%, 14% and 7% for Computer 1, 2 and 3 respectively (Table 1). This shows that it is fortunate to measure all three possible schedules, when OpenMP is applied to a new program, to ensure the best possible optimization. In the preliminary study presented in Appendix A, the *static* schedule appeared to be the best choice. This confirms the theory in Section 2.6.3, where it is stated that the choice of schedule depends on the specific application. The choice also depends on whether or not all iterations require a similar amount of time to be executed.

The schedule which provides best optimization for all computers, is the *dynamic* schedule (Section 2.6.3). Hece this scedule may be regarded as the one best suited for this specific application.

Regardless of the schedule used, parallel CPU computing with OpenMP provides a vast, general optimization of the *Propose* implementation, for all computers tested in this study. The achieved time reduction depends stronger on hardware and OS than it does on the specific OpenMP schedule chosen.

In this section, the simulation time needed for different implementations of *Propose* is compared. All percentage values presented, are calculated from Figure 14 using equation (42) in Section 2.6.1.

The fastest simulation is performed on Computer 1, which by using the fastest OpenMP implementation achieves a 93% reduction in computational time, compared to the original Matlab implementation [2]. Compared to the serial MEX implementation, which is similar to the one presented in my specialization project [1], the same number is 87%.

Running on Computer 2, the serial MEX needs 2% more time to perform the simulation than the original Matlab implementation. The OpenMP implementation however, provides a 77% reduction in computational time, compared to the original Matlab implementation.

Computer 3 showed the best performance obtained for the serial MEX implementation. Optimizing the program using OpenMP resulted in 46% time reduction compared to the serial MEX implementation. Compared to the original Matlab implementation the total reduced time consumption yields 74%.

Figure 14 display the OpenMP simulation times for each of the three possible schedule settings presented in Section 2.6.3. The most important result from these three groups, is that there are very little differences between the various schedule settings. Absolute differences between the different schedules are small, but the relative difference between the best and the worst choice is 21%, 14% and 7% for Computer 1, 2 and 3 respectively (Table 1). This shows that it is fortunate to measure all three possible schedules, when OpenMP is applied to a new program, to ensure the best possible optimization. In the preliminary study presented in Appendix A, the *static* schedule appeared to be the best choice. This confirms the theory in Section 2.6.3, where it is stated that the choice of schedule depends on the specific application. The choice also depends on whether or not all iterations require a similar amount of time to be executed.

The schedule which provides best optimization for all computers, is the *dynamic* schedule (Section 2.6.3). Hece this scedule may be regarded as the one best suited for this specific application.

Regardless of the schedule used, parallel CPU computing with OpenMP provides a vast, general optimization of the *Propose* implementation, for all computers tested in this study. The achieved time reduction depends stronger on hardware and OS than it does on the specific OpenMP schedule chosen.

A comparison between the serial MEX and the Matlab implementation (Figure 14), for Computer 3 in Table 1, shows a 51% reduced computational time. This is very close to the relative results obtained in my specialization project (55%) [1], where Computer 3 was used in all simulations[1]. The relative results from my specialization project [1] may thus be regarded as reproduced for the serial MEX case. The simulation time is reduced by 46% compared to the absolute numbers presented in my specialization project [1]. From the results shown in Figure 14, one also can find that the performance of the serial MEX implementation is varying vastly with the hardware and OS used. For the serial MEX times, Computer 3 shows a value 29% and 52% lower than Computer 1 and 2, respectively. This shows that the results found in my specialization project [1] is not as general the new ones obtained for the OpenMP implementation. This non-generality in the previous results render the serial MEX implementation as an undesirable solution, compared to any OpenMP configuration used. This OS dependency explains the puzzle presented in Section 2.7 and Appendix C, regarding results in my specialization project [1].

The strong OS dependency for the serial MEX case, may be explained by the fact that the MEX file is dynamically linked to all included libraries. Therefore, the OS dependency observed in Figure 14, may indicate a varying efficiency in C++ runtime libraries for the different OSes. At the first glance this seems unlikely, because one will expect such variations to be present also in the OpenMP case. From Figure 14 it is clear that the variations

1 Note that Computer 3 has been updated from 4 Gb to 8 Gb of Random Access Memory (RAM) since the research in [1] was undertaken.

present for the serial MEX case between Computer 2 and 3 is not proportional to the corresponding variations obtained for the OpenMP implementation. However, considering that Computer 2 has eight logical CPU threads while Computer 3 is restricted to four, the variations between OSes might indeed be in accordance for the serial MEX and the OpenMP case. If runtime libraries in Fedora 18 performs computations at half the speed of OS X 10.9.1, but the hardware provides twice as many logical CPUs, it is plausible that the two computers in total perform at a similar level for the OpenMP implementation.

Another explanation might be that the OS dependency is related to how well an OS performs runtime auto-parallelization. The concept of auto-parallelization is briefly presented in [77].

5.2.1 *Simulation time consumption in Abersim and Propose*

Due to the different nature of the simulation methods used in *Propose* and *Abersim* it is hard to compare the simulation times in a precise way. While the quasilinear method simulates a plane directly, *Abersim* uses stepwise propagation meaning that the method depends strongly on the simulation depth. The difference has earlier been commented on in [2], where it was found that the Matlab implementation of *Propose* performed simulations 1000 times faster than *Abersim* for depth below focal depth, and 100 times faster beyond focal depth.

The specific system defined in Section 3.2, simulated for depth $z = 90$ mm, required 31560 s to complete in *Abersim*, and 25 s using the new OpenMP version of *Propose*. This indicates that *Propose* performs this specific simulation approximately 1200 times faster than *Abersim*. These measurements were performed on Computer 1 of Table 1. Note that these comparisons does not include the new C or MEX implementations of *Abersim* as they are not yet possible to install on random computers.

As the *Abersim* software now ships with C and MEX versions, which in the time of writing does not compile out of the box, this is not a strictly fair comparison with respect to *Abersim*. If the *Abersim* team has achieved a similar performance gain by MEX implementation, it is plausible that the relative numbers found in [2] will not have changed, comparing MEX implementations of both methods.

5.3 LIMITATIONS TO THE QUASILINEAR METHOD

The *Propose* method uses quasilinear approximation theory to obtain an estimate of the fundamental and second harmonic field in the case of weak nonlinearity. The method will gradually give less accurate results for propagation in media with strong nonlinear effects, or for high MIs.

Water attenuates higher frequency modes far less efficient than biological tissue. This will cause more energy to diffuse from the fundamental and second order frequency mode, to higher harmonic frequencies. Because of this, the quasilinear method is expected to provide more accurate results in with higher attenuation [26].

In Section 4.1.3 it has been shown, that the *Propose* method is not suitable for describing sharp, rectangular pulses, such as those present close to a rectangular transducer. The reason for this, and implications, has been discussed in Section 5.1.

The *Propose* simulation contains a time-lap at a single depth and not the continues sound field in space such as is the case for *Abersim*. Occasionally one might desire spatial overview of the pulse, and this will in *Propose* require multiple simulations at different depths. *Abersim* already performs stepwise propagation and thus, might be better suited for such a task than *Propose*.

5.4 POSSIBLE SOURCES OF ERROR

The error bar on the highest column in Figure 14 is larger than the others. This may be explained by the fact that Computer 1 in Table 1 is a shared computational server which I did not have exclusive access to, thus it might have been set under additional load during the measuring period. This might also affect the total height of the bar, and thus the relative performance calculated. However, with 2500 samples, the sample standard deviation is expected to define accurate limits for the possible variations.

6 CONCLUSION

In this study, a fast implementation of nonlinear wave simulation in ultrasound, based on the *Propose* method, is presented. The new implementation reduces the required simulation time by 74% – 93%, compared to the original program.

The presented implementation provides a good prediction of the ultrasound field, both with respect to magnitude and phase. This has been found by comparison with the state-of-the-art simulation tool, *Abersim*. Normalized beam profiles are in good agreement for the center lobe and the highest side lobes at simulation depth 70 and 90 mm. This deviation is mainly visible in side lobes below -20 dB and -35 dB, for the fundamental and second harmonic field respectively.

The 90 mm depth simulated in this study, is calculated within seconds using Propose, while the same setup needs more than 12 hours to complete in *Abersim*. This difference is of vast importance for applications like transducer design, where the engineer needs to experiment with various geometries and setups during a work-day. For educational purposes, the fast implementation can provide an efficient method for students to explore responses in the ultrasound pulse, as different parameters are adjusted. *Propose* may also represent an efficient tool for making preliminary simulations, determining different specifications in the setup of other, more accurate and time consuming methods.

The aim of this study has been reached, by providing an implementation which performs simulations within less than 73% of the earlier needed computational time.

SUGGESTIONS FOR FURTHER STUDIES

The simulation of nonlinear wave propagation (*prop2harm*) now is so fast that the bottle neck for large number of simulation points, is in the final Inverse Fast Fourier Transform (IFFT) of the simulated data from Fourier domain to space-time. Thus, a natural step for further optimization of the program, is to write the IFFT function called *fourierToTimeSpace* in a compiled language using the Fastest Fourier Transform in the West (FFTW) or a similar algorithm.

It is also possible to use embedded GPU functions and data types in Matlab's GPU toolkit. This approach might be well suited for the IFFT subroutine *fourierToTimeSpace* in *Propose*, because the Matlab embedded IFFT function is directly available for GPU. With only small amounts of modifications, this subroutine might be ported directly to GPU from Matlab using the *gpuArray* data type. The edited function would still remain compatible with the MEX implementations presented in this study.

The *Propose* software has a text based user interface. To make the method available to a larger audience of scientists, students, and transducer designers, it is suggested to implement a graphical wrapping to the program.

An interesting spin-off subject for future studies is to explore runtime auto-parallelization and compare general performance of various operative systems. This is suggested as an interesting research subject in computational science.

A

OPTIMIZATION RESULTS – *SINC(X)*

A.1 INTRODUCTION AND MOTIVATION

To exploit the general potential in MEX functions for accelerating Matlab® programs, it is interesting on a general basis to compare the performance of a C++ MEX function to a routine that is also built into Matlab as part of the distribution. In this appendix the *sinc(x)* function has been exploited to reveal a the ability for a MEX function to compete with the built in Matlab® version.

In my specialization project [1], where it was found that a non-parallelized C++ MEX version of the *sinc(x)* function was able to compete with the Matlab version. However, a contradictory result was found by Alfonso Rodriguez at the Department of Circulation and Medical Imaging on a Windows® computer (Section 2.7). This contradiction was calling for further exploration of the OS dependency in optimizations utilizing MEX functions, and has thus inspired the current study.

The study can be viewed as an independent preliminary study to search out the necessity for executing performance tests on multiple OSes discussed in Section 3.4. It also provides a "worst-case" environment for testing the optimizing potential related to MEX functions (Section 2.6.2) and OpenMP (Section 2.6.3), as no Matlab functions will be more efficient than its optimized and embedded features[64]. Therefore the study has been important in the preliminary phase of this thesis, to explore whether MEX functions and OpenMP provides a powerful mean to optimize the main implementations in the *Propose* method (Section 3.3).

A.2 THEORY

The mathematical task of the *sinc(x)* function is running through every element of a matrix and computing the normalised sinc(x) value of every element using the formula:

$$\operatorname{sinc}(x) = \begin{cases} 1, & \text{if } x = 0. \\ \frac{\sin(\pi x)}{\pi x}, & \text{otherwise.} \end{cases}$$

Every computation done after this formula is independent of other matrix elements and hence easily lends itself to parallel computation routines. Since it is a precompiled part of the Matlab distribution, it does not suffer from the same performance issues as user provided Matlab code [64].

A.3 METHOD

Measurements were made using tic-toc statements for the various implementations of *sinc*, with 900 measurements for each value and the sample standard deviation as error indicator. The computational time was measured on three different operating systems: Ubuntu 12.04, Fedora 18 and MAC OS X 10.9.1 using Matlab version 2011b [74]. Various OpenMP schedule settings are discussed in Section 2.6.3. The C++ MEX implementation was based on work earlier discussed in [1] and is presented in Section C.3. In the new version, OpenMP has been implemented and the *gsl_complex.h* library has been replaced by the standard *complex.h* library. In the comparison, various schedule settings in OpenMP (Section 2.6.3) are compared to the original MEX file[1], and the built in Matlab function, for a $100 \times 100 \times 100$ matrix.

A.4 RESULTS AND DISCUSSION

The results shown in Fig. 15 indicate that a MEX function will have a hard time competing with a built in Maltab function on a general basis. However it may show better performance if parallelization opportunities are utilised, or the OS has an efficient way of auto parallelizing a serial program at runtime [77].

The results in Figure 15 indicate that Matlab is more easily beaten by a serial C++ MEX function on a low end MacBook®

1 Version 1.0 of the program [1].

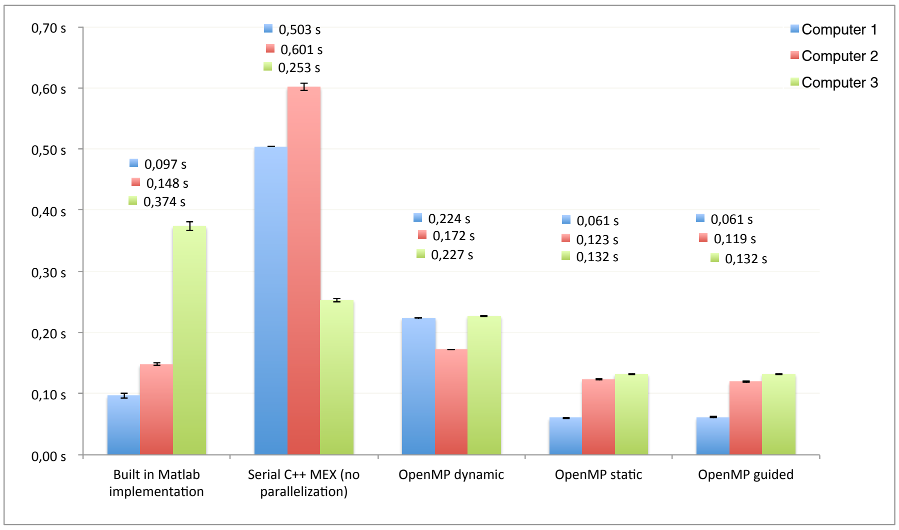

Figure 15: Summary of computational time required for computing the sinc(x) value of every element in a $100 \times 100 \times 100$ matrix for various computers and operating systems. For a thorough discussion of the various OpenMP settings, see Section 2.6.3. The numbers used to denote different computers correspond to Table 1.

.

Pro (Computer 3) than on a high end Linux desktop (Computer 1). The main cause of this seem to be that Matlab runs significantly slower on the MacBook® Pro than on the Linux machines in this study. However, Apple® seem to provide the best efficiency in their auto parallelization algorithm [77] as the serial C++ implementation runs twice as fast there, as on the best corresponding Linux machine[2]. Such a claim needs to be verified using a large sample of various computers in order to be proven, but it proposes an interesting problem for further studies.

Using static or guided OpenMP schedule (Section 2.6.3), we see that performance differences are reduced between Computer 3 and Computer 2. Computer 1 shows the overall highest performance and reaches 1.58 times speed-up compared to the built in function. Computer 2 speeds up 1.24 times while a speed up by 2.83 times is obtained on Computer 1. These results suggest that OpenMP implementations have especially high relative impact on the computational time in the low end MacBook Pro.

2 Even though the fastest Linux desktop has a higher CPU clock frequency.

A.4.1 *Closing comments*

It is important to note that this study provide a worst-case scenario in regard to MEX functions as the Matlab *sinc(x)* function is built into the main software and highly optimized. A user implemented Matlab function will show vastly larger potential for enhancement (Section 4.2).

In this study, a newer Matlab version has been used on Computer 3 than on the other two. To explore the findings further, all machines should be tested using the same version of Matlab. However, it seems unlikely that Matlab has become slower in the last two years. Such a finding would be a much more astonishing result than the one suggested[3].

A.5 CONCLUSION

Compared to the built-in Matlab version, a serial C++ implementation of the *sinc(x)* function does not in general perform faster computations. However, by OpenMP parallelization of the C++ code, all the tested computers display higher performance in the best MEX implementations[4], compared to the built-in Matlab version.

Hence, general optimizing results may be obtained using MEX functions, if the C++ also is parallelized. In this study OpenMP has successfully been used for parallelizing the implementation. The results represent a "worst-case" scenario, as the reference *sinc(x)* function is optimized and embedded in the compiled parts of Matlab. Thus, the method is expected to provide much higher optimizations to a user implemented Matlab function, like the one discussed in Section 3.3.

3 That Matlab perform differently on different OSes.
4 Both the static and guided OpenMP schedules provide general improvements.

B

EFFICIENT DATA ACCESS IN A 3D MATRIX

B.1 INTRODUCTION AND THEORY

This study can be viewed as a preliminary study to the thesis, regarding performance in different for-loop data access. The question in view is whether a single, long for-loop or a nest of shorter loops, will provide the fastest accessing of elements in a 3D matrix.

In a C++ for-loop, a running variable is declared and initialized at the start of the loop. This needs to be done one time if we have a single loop. If we choose to use nested for-loops (Listing 2) the inner ones will have to re-declare their running variable every time the loop outside it finishes an iteration.

Listing 2: A typical triple nested for-loop in C++.

```
for (int k; k < SOME_BORDER_VALUE_K ; k++){
        for (int i; i < SOME_BORDER_VALUE_I ; i++){
                for (int j; j < SOME_BORDER_VALUE_J ; j++){
                (...content...)
                }
        }
}
```

In a MEX file, big chunks of data will often be stored in a Matlab specific data type called *mxArray*. This construct stores the data in the same way as Matlab, where a multidimensional matrix is stored as a one dimensional array no matter how many dimensions it consists of. The dimensionality is stored in internal object variables, and internal functions help a Matlab user to access data in the intuitive *M(i,j,k,...)* manner [68]. This way of storing the data gives the opportunity to ignore the dimensionality altogether if every data element in the matrix is independent of every other element. If one needs the conventional triple indices in calculations or ordering of data, they may be calculated from the major index using the modulo operator in a manner discussed in Section 2.6.4.

In this appendix is examined whether a construct running a single for-loop and the function in Listing 3 can be said to be a more efficient way of accessing a C++ Matlab array of data, than running a triple nested for-loop.

A known benefit from merging a number of nested loops in to one bigger loop is that the catch memory is used in a more efficient manner. In stead of moving through rows and columns, we run though the data in the order it is stored in the memory, making more efficient use of the memory addresses stored in the processor cache. [67, p. 133]

B.2 METHOD

The test has been performed using two versions of the *prop2harm* C++ MEX function discussed in Chapter 3. The reference structure is the original version of *prop2harm* developed in [1], and the other one is the 2.0 version included in App. C.3 of this thesis.

The simulation time of an example matrix were measured both using OpenMP with various parallelization settings and a pure C++ implementation with no OpenMP directives. The computations were performed under using Computer 1 from Table 1, and the simulated system is the one summarized in Table 3.

B.3 RESULTS AND DISCUSSION

In this section, timing results from the implementations in view at two different problem sizes are displayed.

Using (42) on the results displayed in Figure 16. one can find that the implementation using a single for-loop achieves a calculation time reduction by up to 24%, relative to the mean total time needed[1]. This speed-up is achieved under OpenMP and may be related to reduced CPU overhead, as each thread has a more extensive chunk of work containing iterations of various duration. Therefore random differences in computation time for different data points are smoothed out. For the serial implementation, the total numbers show that the old nested loop system is faster by 0.02 s. Relative to the corresponding mean value this number represent a small change $-0,3\%$, but it does

1 For the dynamic OpenMP work flow

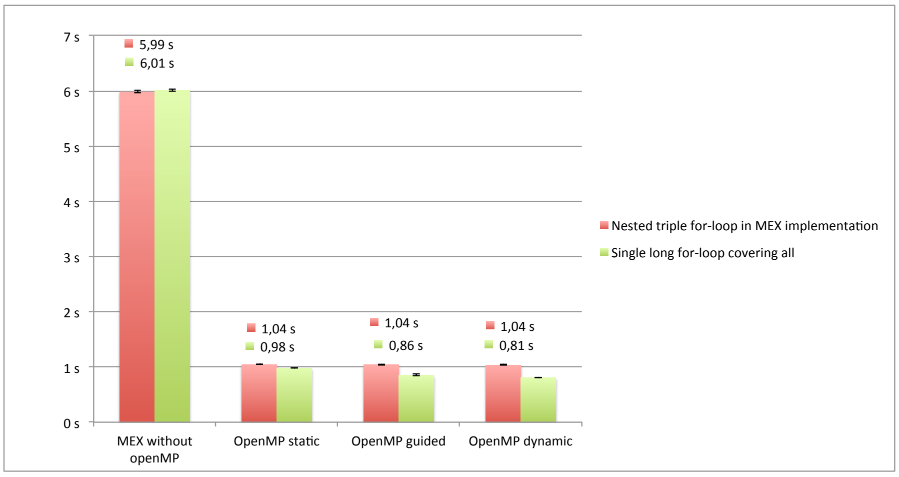

Figure 16: Summary of computational time required for simulation of an ultrasound field in the 2nd harmonic frequency band containing $x \times y \times z = 66 \times 38 \times 11 = 27652$ elements using various OpenMP settings.

show that the speed up achieved in the OpenMP versions of the code is obtained because of reduced overhead and not because of more efficient catch use [67, p. 133], as that also would have given a speed-up for the serial implementation.

B.4 CONCLUSION

In this preliminary study, it was found that parallel processing using OpenMP used 24% less time on a nonlinear simulation when the data access in a large, three dimensional matrix was organized using a single for-loop instead of a nested one. The re-structuring requires that every element of the matrix may be accessed independently. For a serial implementation, a weak negative effect $(-0,3\%)$ is observed from this reordering. However, in relative numbers the difference is too small to defend the use of multiple independent versions of the program, where one is using OpenMP and the other is not.

C

C++ SOURCE CODE

C.1 CONVERSION BETWEEN TRIPLE AND SINGLE INDEXING

Using all zero-based indexing, the subscripts of a matrix may be calculated using the single index, and known dimensional length for the two first indices. The theory of this calculation is explained in Section 2.6.4 and the C++ implementation is displayed in Listing 3. This function makes it possible to exploit the advantages of using a single for-loop and a single index to access elements in the 3D Matlab matrix (App. B), and still have the ability to use the triple subscripts in calculations and element addressing inside the loop.

Listing 3: C++ subfunction used for calculation of one or zero based triple indices from a zero based 3D single index. Part of the MEX function in Listing 5.

```
void tripleIndex(int singleIndex, int indexArray[3], int
    sizeI, int sizeJ, bool baseOfReturnIndex){
    /*The function trippleindex calculates three zero based
        (baseOfReturnIndex=0) or 1-based (baseOfReturnIndex
        =1) indices from a C++ MEX 0-based singleIndex, and
        the 1st and 2nd array lengths.*/
    //Foundational eq. is: singleIndex= i + sizeI*j + sizeI*
        sizeJ*k;
    int IxJ = sizeI * sizeJ;
    int rem = singleIndex % IxJ; //i.e. rem=sizeI*j + i;
    indexArray[0]=( rem % sizeI ); //This is the final i
        value (1st index)
    indexArray[1]=( ( rem - indexArray[0] ) / sizeI ); //
        This is the final j value (2nd index)
    indexArray[2]=( ( singleIndex - indexArray[0] -
        indexArray[1] * sizeI ) / IxJ ); //This is the final
         k value (3rd index)
    if (baseOfReturnIndex){indexArray[0]++; indexArray[1]++;
         indexArray[2]++;}; //Modification of output if 1-
        based return indexing (MATLAB index) is specified (
        defualt return value is 0-based (C++ index))
    return;
};
```

C.2 SOURCE CODE TO THE MEX VERSION OF THE *sinc*(*x*) FUNCTION

The *sinc(x)* MEX function source code displayed in Listing 4 is used in the studies of Appendix A and is based on the version made as part of my specialization project [1]. As part of the present thesis, the following updates have been made:

- The *gsl_complex.h* library has been replaced by the standard C++ *complex.h* library.
- Parallel CPU computing has been implemented using OpenMP directives.
- Minor updates in the structure of the program.

Listing 4: C++ implementation of *sinc* based on the one used in my specialization project [1].

```
//newSinc.cpp v. 2.0 - Haakon Seljaasen//
#include "mex.h"
#include "matrix.h"
#include "math.h"
#include <complex>
using namespace std;

#ifdef _OPENMP
#define ompWORKFLOW guided  //Open MP schedule() setting
#include "omp.h"
#else
    #define omp_get_thread_num() 0
#endif

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[]){
    /*The C++ MEX function "newSinc(x)" takes a matrix as
        input argument and computes the sinc(x)
     *value of any input element sinc(x.). */
//Validation of inputs:
    if (nlhs!=1){
        mexErrMsgIdAndTxt("Toolbox:complexSinc:nlhs","only
            one output is allowed");
    };
    if (nrhs!=1){
        mexErrMsgIdAndTxt("Toolbox:complexSinc:nrhs","only
            one input matrix is allowed");
    };
    //Declaration of variables
    double pi =
        3.1415926535897932384626433832795028841971693993751;
    const mxArray * input=prhs[0];
    complex<double> argument;
    complex<double> I=complex<double>(0.0,1.0);
    double * xInRe=NULL;
    double * xInIm=NULL;
    double * xOutRe=NULL;
    double * xOutIm=NULL;
    bool IsComplex=mxIsComplex(input);
    int totalMatrixSize;
    //Extracting dimmensions from the input Matrix:
    mwSize ndim;
    const mwSize *dims;
    dims=mxGetDimensions(input);
    ndim=mxGetNumberOfDimensions(input);
    totalMatrixSize=mxGetNumberOfElements(input);
    //Creating output Matrix:
    mxClassID classid=mxDOUBLE_CLASS;
    mxComplexity ComplexFlag;
    if (IsComplex){
        ComplexFlag=mxCOMPLEX;
    }else{
        ComplexFlag=mxREAL;
    };
    plhs[0]=mxCreateNumericArray(ndim, dims, classid,
        ComplexFlag);
    //Directing pointers;
    xInRe=mxGetPr(input);
    xOutRe=mxGetPr(plhs[0]);
    if (IsComplex){
        xInIm=mxGetPi(input);
        xOutIm=mxGetPi(plhs[0]);
    };
    //Performing calculations on every element in the Matrix
        :
        if (IsComplex){
```

```
#pragma omp parallel for private(argument) shared(pi,xInRe,
    xInIm,xOutRe,xOutIm) schedule(ompWORKFLOW) //openMP call
            for (int i=0;i<totalMatrixSize;i++){
            if((xInRe[i]==0)&&(xInIm[i]==0)){
                xOutRe[i]=1;//if x=0, sinc(x)=1.
                xOutIm[i]=0;
            }else{
                argument=(xInRe[i]+xInIm[i]*I)*pi;
                argument=sin(argument)/argument;
                xOutRe[i]=real(argument);
                xOutIm[i]=imag(argument);
            };
        };
        }else{
#pragma omp parallel for private(argument) shared(pi,xInRe,
    xInIm,xOutRe,xOutIm) schedule(ompWORKFLOW) //openMP call
            for (int i=0;i<totalMatrixSize;i++){
            if(xInRe[i]==0){
                xOutRe[i]=1;//if x=0, sinc(x)=1.
            }else{
                xOutRe[i]=sin(pi*xInRe[i])/(pi*xInRe[i]);
            };
        };
    };//End parallel computing
    return;
};
```

C.3 SOURCE CODE TO THE MEX VERSION OF *prop2harm*

Version 1.0 of the C++ MEX function used to calculate the second harmonic field (Listing 5) was created as part my specialization project, version 2.0 is a further development of that version [1, App. C.3]. As part of the current research, a number of important updates have been made:

- The *gsl_complex.h* library has been replaced by the standard C++ *complex.h* library.
- To allow for smooth parallel computing, changes have been made in line 198 – 225 by replacing a triple nested for-loop with one single long loop (App. B) utilizing the function in listing.
- Parallel CPU computing has been implemented using OpenMP directives.

- In the Angular Spectrum Method, (22) from Chapter 2 has been implemented on lines 335-339 and 386-390 to reduce approximation errors regarding potential evanescent waves, which may appear close to the transducer.

The original program had an option for utilization of symmetrical properties in simulations, requiring 1/4 of the computations otherwise needed. This possibility is maintained in the new version, but is still organized as a triple nested for-loop as it does not benefit from being converted to a single loop. For clarity reasons the duplicating of values to symmetrical matrix points have been built into the sub-function *utilizeSymmetry*.

Listing 5: Source code for the C++ MEX function used in simulations of the 2nd harmonic field. Based on the implementation used in my specialization project [1].

```
 1 ///prop2harm v. 2.0, part of the Propose simulation program.
        MEX-function written by
   ///master student Haakon Seljaasen fall 2013 (v. 1.0) and
       spring 2014 (v. 2.0)
 3 #include "math.h"
   #include "mex.h"
 5 #include "matrix.h"
   #include "complex.h"
 7 using namespace std;

 9 #define ompWORKFLOW runtime //Open MP schedule() setting -
       affects all calls

11 #ifdef _OPENMP
       #include "headerFiles/omp.h"
13 #else
       #define omp_get_thread_num() 0
15 #endif


17
   //subfunctions:
19 double complex nodePointCalculation(const mxArray * P11,
       const mxArray * K,int fxStart,int fxStop,int fyStart,int
        fyStop,int fStart,int fStop,double z1,double z2,double
       fx,double fy,double f,double a,double b,double c,double
       complex Cfactor,double infinitesimalProduct,int m,int n,
       int l,int NfHalf);
   double complex katet(double fx, double fy, double f, double
       * a, double * b, double c);
```

```
mxArray * katet(const double* fx, int Nfx1, const double* fy, int Nfy1, const double* f, int fLen, const double* a, const double* b, const double * c);
double mean (const double array[], const double length);
double complex csinc (double complex x);
mwIndex singleIndex(int i, int j, int k, int sizeI, int sizeJ, bool baseOfIndex);
void tripleIndex(mwIndex singleIndex, int indexArray[3], int sizeI, int sizeJ, bool baseOfReturnIndex);
void utlizeSymmetry(double complex P221, double * outputRe, double * outputIm,int index[3], int dims[3],int Nx2, int Ny2, bool baseOfInputIndex);

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){
    /*The function prop3harm is computing the 2. harmonic nonlinear propagation from plane P11 to P22, using the
     *quasi-linear approximation. Input parameters are (P11,fx1,fy2,f1,z1,z2,a,b), output parameters are (P22,fx2,fy2,f2).
     * - P11 is the Fouriertransform of the field in pos. z1, with freq. axes fx1,fy1,f1
     * - P22 is the Fouriertransform of the 2. harm field in pos. z2, with freq. axes fx1,fy1,f1.*/
//Tested 29.12.13 and working correct, Haakon Seljaasen.
//Validation of inputs:
    if (nlhs!=4){
        mexErrMsgIdAndTxt("Toolbox:prop2harm:nlhs","exactly 4 output variables is allowed (P22,fx2,fy2,f2)");
    };
    if (nrhs!=8){
        mexErrMsgIdAndTxt("Toolbox:prop2harm:nrhs","exactly 8 inputs are needed: (P11,fx1,fy2,f1,z1,z2,a,b)");
    };
    //Definition of pi:
    long double pi = 3.14159265358979323846264338327950288419716939937510; //(double)(*PI);
    //Calculation method:
    bool useSymmetry;//Assigned from pSettings in Propose
    //Declaration of variables
    const mxArray * P11=prhs[0];
    const mxArray * Z1=prhs[4];
    const mxArray * Z2=prhs[5];
    const double * fx1=mxGetPr(prhs[1]);
    const double * fy1=mxGetPr(prhs[2]);
    const double * f1=mxGetPr(prhs[3]);
    mxArray * Fx1=mxDuplicateArray(prhs[1]);
    mxArray * Fy1=mxDuplicateArray(prhs[2]);
    mxArray * Ff1=mxDuplicateArray(prhs[3]);
    double * fx2;
    double * fy2;
    double * f2;
    double * z1=mxGetPr(prhs[4]);
    double * z2=mxGetPr(prhs[5]);
    double * a=mxGetPr(prhs[6]);
    double * b=mxGetPr(prhs[7]);
    mxArray * A=mxDuplicateArray(prhs[6]);
    mxArray * B=mxDuplicateArray(prhs[7]);
    mxArray * P22;
    mxArray * Fx;
    mxArray * Fy;

    double * P22Re=NULL;
    double * P22Im=NULL;
    //Various complex variables:
    double complex Kf;
    double complex P221;
    double complex ImgPropFactor;
    //Various framework variables:
    double temp=0;
    int Nfx1, Nfy1, Nf1, Nx, Ny, Nf, Nx2, Ny2, Nf2, NfHalf,
        Nl, Nm, Nn;
    int fxStart,fxStop,fyStart,fyStop,fStart,fStop;
    double dfx, dfy, df, f0, limit, f, fy, fx;
    mxArray * K;
    mwIndex sizeOfMatrix;
    int oneBasedIndex[3];
    double kappa, betaN, c;//Acoustical properties
    //Extracting setting values from the global workspace,
        pSettings.
    //
        -----------------------------------------------------------

    mxArray *pSettings=mexGetVariable("global", "pSettings")
        ; // get the struct
    if (pSettings==NULL){//error testing
        //mexPrintf("\nError in prop2harm.cpp: Struct
            pSettings not found\n");
        mexErrMsgIdAndTxt("Toolbox:prop2harm:pSettings","Matlab variable pSettings not found");
    };
    //------------------------useSymmetry?-------------------------------
    mxArray *Symmetry = mxGetField(pSettings,0,"useSymmetry"); // get the field
    if (Symmetry==NULL){//error testing
        mexErrMsgIdAndTxt("Toolbox:prop2harm:Symmetry","Struct value pSettings.useSymmetry not found");
    };
    if ((*mxGetPr(Symmetry)==0.0)||(*mxGetPr(Symmetry)==1)){//error testing
        useSymmetry=(bool)(*mxGetPr(Symmetry));
    }else{
        mexErrMsgIdAndTxt("Toolbox:prop2harm:Symmetry","Struct value pSettings.useSymmetry contains non-boolean value");
    };

    mxArray *Kappa = mxGetField(pSettings,0,"kappa"); // get the field
    if (Kappa==NULL){//error testing
        mexErrMsgIdAndTxt("Toolbox:prop2harm:Kappa","Struct value pSettings.Kappa not found");
    };
    kappa=(*mxGetPr(Kappa));
    mxArray *BetaN = mxGetField(pSettings,0,"betaN"); // get the field
    if (BetaN==NULL){//error testing
        mexErrMsgIdAndTxt("Toolbox:prop2harm:BetaN","Struct value pSettings.BetaN not found");
    };
    betaN=(*mxGetPr(BetaN));
    mxArray *SpeedOfSound = mxGetField(pSettings,0,"c"); // get the field
    if (SpeedOfSound==NULL){//error testing
        mexErrMsgIdAndTxt("Toolbox:prop2harm:SpeedOfSound","Struct value pSettings.c not found");
    };
    c=*(mxGetPr(SpeedOfSound));
    ImgPropFactor=0-(kappa*betaN*pow(pi,2)/pow(c,4))*I;
    //Finding properties of input Matrix.
    Nfx1=mxGetN(prhs[1]);
    Nfy1=mxGetN(prhs[2]);
    Nf1=mxGetN(prhs[3]);
    mwSize ndim=3; //3D Matrix output
    //Extracting dimmensions from the input Matrix:
    const mwSize * P11Dims=mxGetDimensions(P11);
    Nx=P11Dims[0];
    Ny=P11Dims[1];
    Nf=P11Dims[2];
    mwIndex tempIndex[3];
    //Allocating mxArrays and pointers for output:
    tempIndex[0]=1;     //Needed to make the output Arrays
       column major.
    tempIndex[1]=2*Nx;  //using this temp. variable to
       satisfy input requirements in mxCreateNumericArray.
    plhs[1]=mxCreateNumericArray(2, tempIndex,
       mxDOUBLE_CLASS, mxREAL);//Allocating output mxArray
    fx2=mxGetPr(plhs[1]);//Making double pointer to output,
       for use inside prop2harm.
    tempIndex[1]=2*Ny;
    plhs[2]=mxCreateNumericArray(2, tempIndex,
       mxDOUBLE_CLASS, mxREAL);
    fy2=mxGetPr(plhs[2]);//Making double pointer to output,
       for use inside prop2harm.
    tempIndex[1]=Nf;
    plhs[3]=mxCreateNumericArray(2, tempIndex,
       mxDOUBLE_CLASS, mxREAL);
    f2=mxGetPr(plhs[3]);//Making double pointer to output,
       for use inside prop2harm.
    dfx=fx1[1]-fx1[0];
    dfy=fy1[1]-fy1[0];
    df=f1[1]-f1[0];
    double infinitesimalProduct=dfx*dfy*df;
    //Frequency axes for 2. harmonic:
    f0=mean(f1,Nf1);
    Nf2=Nf1;
    for(int i=0;i<Nf1;i++){
        f2[i]=f1[i]+f0;
    };
//Initializing counters for determination of array length
   Nx2 and Ny2:
    Nx2=0;
    temp=2*fx1[0];
    limit=2*fx1[Nfx1-1]+dfx;
    while(temp<limit){
        fx2[Nx2]=2*fx1[0]+dfx*(Nx2);
        temp=fx2[Nx2];
        //mexPrintf("\nfx2=%f\n",temp);
        Nx2++;
```

```
};
Ny2=0;
temp=2*fy1[0];
limit=2*fy1[Nfy1-1]+dfy;
while(temp<limit){
    fy2[Ny2]=2*fy1[0]+dfy*(Ny2);
    temp=fy2[Ny2];
    //mexPrintf("\nfy2=%f\n",temp);
    Ny2++;
};
//---------------------------------------------------
/*If one wish to include dispersion this can be included
     here,
 * this function may be called from matlab: cw=
     dispersion(f2,2*f0,a,b)*/
//Creating output Matrix:
mwSize P22Dims[3];
P22Dims[0]=Nx2;
P22Dims[1]=Ny2;
P22Dims[2]=Nf2;
plhs[0]=mxCreateNumericArray(ndim, P22Dims,
    mxDOUBLE_CLASS, mxCOMPLEX);
P22=plhs[0];
sizeOfMatrix=mxGetNumberOfElements(P22);
//Directing return Matrix pointers;
P22Re=mxGetPr(P22);
P22Im=mxGetPi(P22);
//Making Matrix/table with K values for use in the
    nodePointCalculation function:
K=katet(fx1, Nfx1, fy1, Nfy1, f1 , Nf1, a, b, &c); //
    Calculating katet() values for every combination of
    fx, fy and f...
//Boundaries for the tripleIndecies
NfHalf=floor(Nf/2.0);
Nl=Nf;
if(useSymmetry){
    Nm=round(Nx2/2.0);
    Nn=round(Ny2/2.0);
}else{
    Nm=Nx2;
    Nn=Ny2;
};

//Main calculations:
if ( ! useSymmetry ){
```

```
#pragma omp parallel for private(fy,fx,f,fyStart,fyStop,
    fxStart,fxStop,fStart,fStop,P221,oneBasedIndex) shared(
    P22Dims,P11,K,z1,z2,a,b,c,ImgPropFactor,
    infinitesimalProduct,NfHalf,sizeOfMatrix) schedule(
    ompWORKFLOW) //Calling openMP
        for (mwIndex t=0; t<sizeOfMatrix; t++){

            tripleIndex(t, oneBasedIndex, P22Dims[0],
                P22Dims[1], 1);
//The oneBasedIndex values corresponds to the following
    indecies used in the original MATLAB code:
//      m=oneBasedIndex[0]; //x-dimension
//      n=oneBasedIndex[1]; //y-dimension
//      l=oneBasedIndex[2]; //f-dimansion
            fx=fx2[oneBasedIndex[0]-1];
            fxStart=fmax(1,1+oneBasedIndex[0]-Nx);
            fxStop=fmin(Nx-1,oneBasedIndex[0]);

            fy=fy2[oneBasedIndex[1]-1];
            fyStart=fmax(1,1+oneBasedIndex[1]-Ny);
            fyStop=fmin(Ny-1,oneBasedIndex[1]);

            f=f2[oneBasedIndex[2]-1];
            fStart=fmax(1,1+oneBasedIndex[2]-NfHalf);
            fStop=fmin(Nf,oneBasedIndex[2]-NfHalf+Nf-1);
            if  (!((fStart>fStop)||(fyStart>fyStop)||(
                fxStart>fxStop))){
                P221=nodePointCalculation(P11,K,fxStart,
                    fxStop,fyStart,fyStop,fStart,fStop,*z1,*
                    z2,fx,fy,f,*a,*b,c,ImgPropFactor,
                    infinitesimalProduct,oneBasedIndex[0],
                    oneBasedIndex[1],oneBasedIndex[2],NfHalf
                    );
            }else{
                P221=0+0*I;
            };
            P22Re[t]=creal(P221);
            P22Im[t]=cimag(P221);
        }; // end of for (t=0 to sizeOfMatrix)
    }else{
//----------------------if symmetry is enabled
    ---------------------------
#pragma omp parallel for private(fy,fx,f,fyStart,fyStop,
    fxStart,fxStop,fStart,fStop,P221,oneBasedIndex) shared(
    P22Dims,P11,K,z1,z2,a,b,c,ImgPropFactor,
    infinitesimalProduct,NfHalf) schedule(ompWORKFLOW) //
    Calling openMP
        for (int l=1; l<=Nl; l++){
            f=f2[l-1];
            fStart=fmax(1,1+l-NfHalf);
            fStop=fmin(Nf,l-NfHalf+Nf-1);
            for (int m=1;m<=Nm;m++){
                fx=fx2[m-1];
                fxStart=fmax(1,1+m-Nx);
                fxStop=fmin(Nx-1,m);
                for (int n=1;n<=Nn;n++){
                    fy=fy2[n-1];
                    fyStart=fmax(1,1+n-Ny);
                    fyStop=fmin(Ny-1,n);
                    if  (!((fStart>fStop)||(fyStart>fyStop)
                        ||(fxStart>fxStop))){
                        P221=nodePointCalculation(P11,K,
                            fxStart,fxStop,fyStart,fyStop,
                            fStart,fStop,*z1,*z2,fx,fy,f,*a
                            ,*b,c,ImgPropFactor,
                            infinitesimalProduct,m,n,l,
                            NfHalf);
                    }else{
                        P221=0+0*I;
                    };
                    oneBasedIndex[0]=m;
                    oneBasedIndex[1]=n;
                    oneBasedIndex[2]=l;
                    utlizeSymmetry( P221,P22Re,P22Im,
                        oneBasedIndex,P22Dims,Nx2,Ny2,1 );
                };//for m=0 to Nn
            }; //for l=0  to Nm
        }; //for l=0 to Nl
    };//end of if-else statement regarding symmetry

    return;
};


double complex nodePointCalculation(const mxArray * P11,
    const mxArray * K,int fxStart,int fxStop,int fyStart,int
     fyStop,int fStart,int fStop,double z1,double z2,double
    fx,double fy,double f,double a,double b,double c,double
    complex Cfactor,double infinitesimalProduct,int m,int n,
    int l,int NfHalf){
```

```
    /*The function nodePointCalculation is used for
       computing the 2. harmonic nonlinear soundfield of a
       node in P22 of prop2harm, using the
     *quasi-linear approximation. Input parameters are (P11,
        K,kf,fxStart,fxStop,fyStart,fyStop,fStart,fStop,z1,
        z2,fx,fy,f,a,b,c,Cfactor,dfx,dfy,df,l,m,n,NfHalf)
        */
//Tested 29.12.13 and working correct, HS.
    double pi =
       3.1415926535897932384626433832795028841971693993751;
    //Output:
    double complex Node=0+0*I;
    //Input
    double * P11Re=mxGetPr(P11);
    double * P11Im=NULL;
    bool complexP11=mxIsComplex(P11);
    if (complexP11){
        P11Im=mxGetPi(P11);
    }else{
        mexErrMsgIdAndTxt("Toolbox:prop2harm:
           nodePointCalculation","Non-complex P11 input
           given, needs to be taken into account in the
           prop2harm.cpp code before enabeling.");
    };
    bool complexK=mxIsComplex(K);
    double * K_Real=mxGetPr(K);
    double * K_Imag=NULL;
    if (complexK){
        K_Imag=mxGetPi(K);//Taken into account later in the
           code with the complexK-variable
    };
    double complex kf=katet(fx, fy, f, &a, &b, c);
    const mwSize * Kdim=mxGetDimensions(K);
    const mwSize * P11dim=mxGetDimensions(P11);
    //Working variables
    double complex gamma;
    double complex K1;
    double complex K2;
    double complex P11A;
    double complex P11B;
    double Z1minusZ2_Half=((z1)-(z2)/2);//for use in Hp
    double MinusZ2div2pi=-(z2)/(2*pi);//for use in Hp
    mwIndex singleIndex1;
    mwIndex singleIndex2;
    mwIndex Ix2, Iy2, If2;
    double complex Hp;
    //Calculations
    for (int h=fxStart; h<=fxStop; h++){
        Ix2=1+m-h;
        for(int i=fyStart; i<=fyStop; i++){
            Iy2=1+n-i;
            for(int j=fStart; j<=fStop; j++){
                If2=l+1+NfHalf-j;
                //Calculating single index
                singleIndex1=singleIndex(h, i, j, Kdim[0],
                    Kdim[1],1);
                singleIndex2=singleIndex(Ix2, Iy2, If2, Kdim
                    [0], Kdim[1],1);
                if(complexK){
                    K1=K_Real[singleIndex1]+K_Imag[
                        singleIndex1]*I;
                    K2=K_Real[singleIndex2]+K_Imag[
                        singleIndex2]*I;
                }else{
                    K1=K_Real[singleIndex1]+0*I;
                    K2=K_Real[singleIndex2]+0*I;
                };
                gamma=K1+K2-kf;
                Hp=cexp(gamma*I*Z1minusZ2_Half)*csinc(gamma*
                    MinusZ2div2pi);
                //Recalculating single index
                singleIndex1=singleIndex(h, i, j, P11dim[0],
                     P11dim[1],1);
                singleIndex2=singleIndex(Ix2, Iy2, If2,
                    P11dim[0], P11dim[1],1);
                P11A=P11Re[singleIndex1]+P11Im[singleIndex1
                    ]*I;
                P11B=P11Re[singleIndex2]+P11Im[singleIndex2
                    ]*I;
                Node=Node+(P11A*P11B*Hp);
            };
        };
    };
    Node=Node*cexp( ( -kf*I )*( z2-z1 ) )*z2*pow( f,2 )*
        infinitesimalProduct;
    Node=Node*( Cfactor / ( kf+2*pi*f/c ) );
    return Node;
};

double complex katet(double fx, double fy, double f, double
    * a, double * b, double c){
```

```
    /*The function katet is ment for computing sqrt(f^2-f_x
        ^2-f_y^2). Input is
     * (fx[],fy[],f[],a,b,c) Output is double a scalar that
         is negative if the answer is complex. a and b are
         not used unless we want to include attenuation in
         the formula.*/
//Definition of pi:
    double pi =
        3.14159265358979323846264338327950288419716939937 51;
    int int_b=*b;
    double complex K;
    double arg=pow(f,2)-pow(fx,2)-pow(fy,2);
    if (arg<0) {
        K=( -csqrt(arg)+f )*( (2*pi)/(c) );
    }else{
        K=( csqrt(arg)-f )*( (2*pi)/(c) );
    };
    //Subtracting attenuation:
    K=K-(*a)*pow(f/pow(10.0,6),int_b)*I;
    return K;
};

//katet.cpp - Haakon Seljaasen//
mxArray * katet(const double* fx, int Nfx1, const double* fy
    , int Nfy1, const double* f, int fLen, const double* a,
    const double* b, const double * c){
    /*The function katet is ment for computing sqrt(f^2-f_x
        ^2-f_y^2). Input is
     * (fx[],Nfx,fy[],Nfy,f[],Nf,a,b,c)*/
//Definition of pi:
    double pi =
        3.14159265358979323846264338327950288419716939937 51;
//Declaration of variables
    mxArray * output;
    double * sqrtRe=NULL;
    double * sqrtIm=NULL;
    int int_b=*b;
    double proportionalityFactor=(2*pi)/(*c);
    double arg;
    double complex K;
//Declaring dimmensions on the output array.
    mwSize ndim=3;
    mwSize dims[3];
//Creating output Matrix
//Set dimmensions:
    dims[0]=Nfx1;
    dims[1]=Nfy1;
    dims[2]=fLen;
    mxClassID classid=mxDOUBLE_CLASS;
    mxComplexity ComplexFlag=mxCOMPLEX;
    mwIndex It;
    if((Nfx1==1)&&(Nfy1==1)&&(fLen)){
        output=mxCreateDoubleMatrix(1, 1, mxCOMPLEX);
    }else{
        output=mxCreateNumericArray(ndim, dims, classid,
           ComplexFlag);
        sqrtIm=mxGetPi(output);
    };
    sqrtRe=mxGetPr(output);
//    #pragma omp parallel for private(arg,root,K,
    attenuation,It) schedule(ompWORKFLOW) //Usually too
    small matrixes to make any positive difference
    //The function is only called once.
    for(int i=0;i<fLen;i++){
        for(int j=0;j<Nfx1;j++){
            for(int k=0;k<Nfy1;k++){
                /* According to original matlab souce code
                   here we make a change
                 * to make sure that the imaginary part of K
                     is negative, so that the solution
                 * does not diverge as z approaches infinity
                     . */
                arg=pow(f[i],2)-pow(fx[j],2)-pow(fy[k],2);
                if (arg<0) {
                    K=(-csqrt(arg)+f[i])*
                        proportionalityFactor;
                }else{
                    K=(csqrt(arg)-f[i])*
                        proportionalityFactor;
                };
                //Accounting for power law attenuation:
                K=K-((*a)*pow(f[i]/pow(10.0,6),int_b))*I;
                //Shipping out results:
                It=singleIndex(j, k, i, dims[0], dims[1],0);
                sqrtRe[It]=creal(K);
                sqrtIm[It]=cimag(K);
            };
        };
    };
    return output;
};
```

```
double complex csinc (double complex x){
    //Definition of pi:
    double pi =
        3.1415926535897932384626433832795028841971693993751;
    if((creal(x)==0.0)&&(cimag(x)==0.0)){
        return 1+0*I;
    }else{
        x=x*pi;//Normalized sinc uses x times pi as argument
        return csin(x)/x;
    };
};

double mean (const double array[], const double length){
    double sum = 0;
    for (int i = 0; i < length; i++){
        sum =sum+ array [i];
    }
    return sum/length;
};

mwIndex singleIndex(int i, int j, int k, int sizeI, int
    sizeJ, bool baseOfIndex){
    /*Creates a 0-based single subscript from the 1-based
        matlab indexi of a 3D matrix
     * if baseOfIndex==1, and a 0 base index of 0 based
         indices if baseOfIndex==0 indices of a 3D matrix.
     * i, j, k are indices wished to access, I and J denotes
         the first two dimmensions of the 3D Matrix*/
    if(baseOfIndex){
        return (i-1)+(j-1)*sizeI+(k-1)*sizeI*sizeJ;
    }else{
        return (i)+(j)*sizeI+(k)*sizeI*sizeJ;
    };
};

void tripleIndex(mwIndex singleIndex, int indexArray[3], int
     sizeI, int sizeJ, bool baseOfReturnIndex){
    //This function is displayed separate in Listing 3.
    };


void utlizeSymmetry(double complex P221, double outputRe[],
    double outputIm[],int index[3], int dims[3],int Nx2, int
     Ny2, bool baseOfInputIndex){
    ///This function is used to distribute a computed value
        in prop2harm to all 4 destinations in the output
        array where it applies according to symmetry
        properties.
    mwIndex tempIt;
    tempIt=singleIndex(index[0], index[1], index[2], dims
        [0], dims[1], baseOfInputIndex);
    outputRe[tempIt]=creal(P221);
    outputIm[tempIt]=cimag(P221);

    tempIt=singleIndex(Nx2-index[0]+1, index[1], index[2],
        dims[0], dims[1], baseOfInputIndex);
    outputRe[tempIt]=creal(P221);
    outputIm[tempIt]=cimag(P221);

    tempIt=singleIndex(Nx2-index[0]+1, Ny2-index[1]+1, index
        [2], dims[0], dims[1], baseOfInputIndex);
    outputRe[tempIt]=creal(P221);
    outputIm[tempIt]=cimag(P221);

    tempIt=singleIndex(index[0], Ny2-index[1]+1, index[2],
        dims[0], dims[1], baseOfInputIndex);
    outputRe[tempIt]=creal(P221);
    outputIm[tempIt]=cimag(P221);
    return;
};
```

BIBLIOGRAPHY

[1] H. Seljåsen, "Fast simulation of nonlinear wave propagation in medical ultrasound," 1 2014. Specialization project, Norwegian University of Science and Technology. http://folk.ntnu.no/seljasen/Arbeider/Specialization_project_2013.pdf. Accessed: 2013-03-04.

[2] F. Prieur, T. F. Johansen, S. Holm, and H. Torp, "Fast simulation of second harmonic ultrasound field using a quasi-linear method," *J. Acoust. Soc. Am.*, vol. 131, no. 6, pp. 4365–4375, 2012.

[3] S. N. Gurbatov, O. V. Rudenko, and A. Saichev, *Waves and Structures in Nonlinear Nondispersive Media: General Theory and Applications to Nonlinear Acoustics*. Springer, 2012.

[4] F. A. Duck, "Nonlinear acoustics in diagnostic ultrasound," *Ultrasound in medicine & biology*, vol. 28, no. 1, pp. 1–18, 2002.

[5] P. H. Torp, "Guest lecture on medical signal processing." TTT4120 – Digital Signal Processing. The The Norwegian University of Science and Technology, November, 14 2013.

[6] R. Hansen, S.-E. Måsøy, T. A. Tangen, and B. A. Angelsen, "Nonlinear propagation delay and pulse distortion resulting from dual frequency band transmit pulse complexes," *The Journal of the Acoustical Society of America*, vol. 129, no. 2, pp. 1117–1127, 2011.

[7] R. S. Shapiro, A. Stancato-Pasik, and S. E. Sims, "Diagnostic value of tissue harmonic imaging compared with conventional sonography," *Computers in biology and medicine*, vol. 35, no. 8, pp. 725–733, 2005.

[8] T. S. Desser, T. Jedrzejewicz, and C. Bradley, "Native tissue harmonic imaging: Basic principles and clinical applications," *Ultrasound Quarterly*, vol. 16, no. 1, pp. 40–hyhen, 2000.

[9] G. A. Whalley, G. D. Gamble, H. J. Walsh, N. Sharpe, and R. N. Doughty, "Quantitative evaluation of regional endocardial visualisation with second harmonic imaging and contrast left ventricular opacification in heart failure patients," *European Journal of Echocardiography*, vol. 6, no. 2, pp. 134–143, 2005.

[10] G. A. Whalley, G. D. Gamble, H. J. Walsh, S. P. Wright, S. Agewall, N. Sharpe, and R. N. Doughty, "Effect of tissue harmonic imaging and contrast upon between observer and test-retest reproducibility of left ventricular ejection fraction measurement in patients with heart failure," *Eur. J. Heart Failure*, no. 6, pp. 85–93, 2004.

[11] C.-L. Yen, C.-M. Jeng, and S.-S. Yang, "The benefits of comparing conventional sonography, real-time spatial compound sonography, tissue harmonic sonography, and tissue harmonic compound sonography of hepatic lesions," *Clinical imaging*, vol. 32, no. 1, pp. 11–15, 2008.

[12] S. Tanaka, O. Oshikawa, T. Sasaki, T. Ioka, and H. Tsukuma, "Evaluation of tissue harmonic imaging for the diagnosis of focal liver lesions," *Ultrasound in medicine & biology*, vol. 26, no. 2, pp. 183–187, 2000.

[13] T. Schmidt, C. Hohl, P. Haage, M. Blaum, D. Honnef, C. Weiβ, G. Staatz, and R. Gunther, "Diagnostic accuracy of phase-inversion tissue harmonic imaging versus fundamental b-mode sonography in the evaluation of focal lesions of the kidney," *American Journal of Roentgenology*, vol. 180, no. 6, pp. 1639–1647, 2003.

[14] A. Van der Steen, J. Poulsen, E. Cherin, and F. Foster, "Harmonic imaging at high frequencies for ivus," in *Ultrasonics Symposium, 1999. Proceedings. 1999 IEEE*, vol. 2, pp. 1537–1540, IEEE, 1999.

[15] H. Becher, K. Tiemann, T. Schlosser, C. Pohl, N. C. Nanda, M. A. Averkiou, J. Powers, and B. Lüderitz, "Improvement in endocardial border delineation using tissue harmonic imaging," *Echocardiography*, no. 15, pp. 511–517, 1998.

[16] R. J. Graham, W. Gallas, J. S. Gelman, L. Donelan, and R. E. Peverill, "An assessment of tissue harmonic versus

fundamental imaging modes for echocardiographic measurements," *J Am Soc Echocardiography*, no. 14, pp. 1191–1196, 2001.

[17] J. H. Ginsberg and M. F. Hamilton, "Computational methods," in *Nonlinear Acoustics* (M. F. Hamilton and D. T. Blackstock, eds.), pp. 309–341, San Diego: Academic press, 1998.

[18] Y.-S. Lee, M. F. Hamilton, and R. Cleveland, "Kzktexas." Accessed January 4, 2014.

[19] S. I. Aanonsen, T. Barkve, J. N. Tjo, *et al.*, "Distortion and harmonic generation in the nearfield of a finite amplitude sound beam," *The Journal of the Acoustical Society of America*, vol. 75, no. 3, pp. 749–768, 1984.

[20] J. Berntsen, "Numerical calculations of finite amplitude sound beams," in *Frontiers of Nonlinear Acoustics: Proceedings of 12th ISNA* (M.F.Hamilton and D.T.Blackstock, eds.), pp. 191–196, Elsevier Applied Science, 1990.

[21] M. E. Frijlink, H. Kaupang, T. Varslot, and S.-E. Masoy, "Abersim: a simulation program for 3d nonlinear acoustic wave propagation for arbitrary pulses and arbitrary transducer geometries," in *Ultrasonics Symposium, 2008. IUS 2008. IEEE*, pp. 1282–1285, IEEE, 2008.

[22] T. Varslot and S.-E. Masøy, "Forward propagation of acoustic pressure pulses in 3d soft biological tissue," *Modeling, Identification and Control*, vol. 27, no. 3, pp. 181–200, 2006.

[23] Y. Du, H. Jensen, and J. A. Jensen, "Angular spectrum approach for fast simulation of pulsed non-linear ultrasound fields," in *Ultrasonics Symposium (IUS), 2011 IEEE International*, pp. 1583–1586, IEEE, 2011.

[24] F. Varray, C. Cachard, A. Ramalli, P. Tortoli, and O. Basset, "Simulation of ultrasound nonlinear propagation on GPU using a generalized angular spectrum method," *EURASIP Journal on Image and Video Processing*, vol. 2011, no. 1, p. 17, 2011.

[25] J. H. Ginsberg, "Model equations," in *Perturbation Methods* (M. F. Hamilton and D. T. Blackstock, eds.), pp. 279–308, San Diego: Academic press, 1998.

[26] S. Dursun, T. Varslot, T. Johansen, B. Angelsen, and H. Torp, "Fast 3d simulation of 2nd harmonic ultrasound field from arbitrary transducer geometries," *IEEE Ultrasonics Symposium*, vol. 2, pp. 1964–1967, 2005.

[27] H. Torp, T. F. Johannsen, and J. S. Haugen, "Nonlinear wave propagation – A fast 3D simulation method based on quasi-linear approximation of the second harmonic field," in *Proceedings of the IEEE Ultrasonics Symposium 2002*, vol. 1, (Munich, Germany), pp. 567–570, 2002.

[28] MATLAB®, *Version 8.2.0.701 (R2013b)*. The MathWorks Inc., Natrick, MA, August 2013. Software.

[29] M. F. Hamilton and C. L. Morfey, "Model equations," in *Nonlinear Acoustics* (M. F. Hamilton and D. T. Blackstock, eds.), pp. 41–64, San Diego: Academic press, 1998.

[30] E. A. Zabolotskaya and R. V. Khokhlov, "Quasi-plane waves in the nonlinear acoustics of confined beams," *Sov. Phys. Acoust*, vol. 15, no. 1, pp. 35–40, 1969.

[31] V. P. KUZNETSO, "Equations of nonlinear acoustics," *SOVIET PHYSICS ACOUSTICS-USSR*, vol. 16, no. 4, p. 467, 1971.

[32] M. F. Hamilton, J. N. Tjo, *et al.*, "Nonlinear effects in the farfield of a directive sound source," *The Journal of the Acoustical Society of America*, vol. 78, no. 1, pp. 202–216, 1985.

[33] P. T. Christopher and K. J. Parker, "New approaches to the linear propagation of acoustic fields," *The Journal of the Acoustical Society of America*, vol. 90, no. 1, pp. 507–521, 1991.

[34] H. F. Johnson, "An improved method for computing a discrete hankel transform," *Computer physics communications*, vol. 43, no. 2, pp. 181–202, 1987.

[35] N. Bakhvalov, Y. M. Zhileikin, E. Zabolotskaya, and R. T. Beyer, *Nonlinear theory of sound beams*. American Institute of Physics Melville, 1987.

[36] K. Frøysa, J. N. Tjøtta, and J. Berntsen, "Finite amplitude effects in sound beams. pure tone and pulsed excitation," *Advances in nonlinear acoustics*, pp. 233–238, 1993.

[37] R. T. Beyer, "The parameter b/a," in *Nonlinear Acoustics* (M. F. Hamilton and D. T. Blackstock, eds.), pp. 25–39, San Diego: Academic press, 1998.

[38] J. A. Jensen, "Field: A program for simulating ultrasound systems," in *10TH NORDICBALTIC CONFERENCE ON BIOMEDICAL IMAGING, VOL. 4, SUPPLEMENT 1, PART 1: 351–353*, Citeseer, 1996.

[39] T. Varslot, S. Dursun, T. Johansen, R. Hansen, B. Angelsen, and H. Torp, "Influence of acoustic intensity on the second-harmonic beam-profile," in *Ultrasonics, 2003 IEEE Symposium on*, vol. 2, pp. 1847–1850, IEEE, 2003.

[40] T. Varslot, G. Taraldsen, T. Johansen, and B. A. J. Angelsen, "Computer simulation of forward wave propagation in non-linear, heterogeneous, absorbing tissue," in *Ultrasonics Symposium, 2001 IEEE*, vol. 2, pp. 1193–1196 vol.2, 2001.

[41] M. O'Donnell, E. Jaynes, and J. Miller, "Kramers–kronig relationship between ultrasonic attenuation and phase velocity," *The Journal of the Acoustical Society of America*, vol. 69, no. 3, pp. 696–701, 1981.

[42] P. A. Westervelt, "Parametric acoustic array," *J. Acoust. Soc. Am.*, vol. 35, pp. 535–537, 1963.

[43] G. Taraldsen, "A generalized westervelt equation for nonlinear medical ultrasound," *The Journal of the Acoustical Society of America*, vol. 109, no. 4, pp. 1329–1333, 2001.

[44] H. Kaupang, "Abersim 2.x – reference manual with tutorials," 2008. Abersim: Users manual. http://folk.ntnu.no/sveinmas/Abersim/abersim2_manual.pdf. Accessed: 2013-04-19.

[45] A. Bamberger, B. Engquist, L. Halpern, and P. Joly, "Parabolic wave equation approximations in heterogenous media," *SIAM Journal on Applied Mathematics*, vol. 48, no. 1, pp. 99–128, 1988.

[46] E. G. Williams, *Fourier Acoustics*, ch. 2.9, pp. 31–32. London: Academic Press, 1999.

[47] B. r. Angelsen, *Ultrasound imaging: Waves, signals, and signal processing*, vol. 2. Emantec, 2000.

[48] K. Rottmann, *Matematisk formelsamling*, vol. 11. Spektrum forlag, 2003.

[49] D. Belgroune, J. de Belleval, and H. Djelouah, "Modelling of the ultrasonic field by the angular spectrum method in presence of interface," *Ultrasonics*, vol. 40, no. 1, pp. 297–302, 2002.

[50] Free Software Foundation, Inc, "GNU General Public License," *Boston, USA, version*, vol. 3, 2007.

[51] B. Angelsen, M. Frijlink, T. F. Johansen, H. Kaupang, R. Hansen, S.-E. Måsøy, P. Näsholm, T. A. Tangen, G. Taraldsen, and T. Varslot, "Abersim simulation software – download page," 10 2009. http://www.ntnu.edu/isb/ultrasound/abersim/download. Accessed: 2013-04-19.

[52] I. Kappel, "Evolution equations and approximations," 1997.

[53] J. D. Dollard and C. N. Friedman, "Product integration with applications to differential equations," 1979.

[54] S. K. Godunov, "A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics," *Matematicheskii Sbornik*, vol. 89, no. 3, pp. 271–306, 1959. Translated by US Joint Publ. Res. Service, JPRS 7226, 1969.

[55] G. Strang, "On the construction and comparison of difference schemes," *SIAM Journal on Numerical Analysis*, vol. 5, no. 3, pp. 506–517, 1968.

[56] M. B. Abbott, *An introduction to the method of characteristics*. New York, NY: American Elisever Publishing Company, Inc., 1966.

[57] F. A. Duck, *Physical properties of tissue: a comprehensive reference book*. San Diego, CA: Academic Press, 1990.

[58] T. L. Szabo, "Time domain wave equations for lossy media obeying a frequency power law," *The Journal of the Acoustical Society of America*, vol. 96, no. 1, pp. 491–500, 1994.

[59] M. F. Hamilton, "Spund beams," in *Nonlinear Acoustics* (M. F. Hamilton and D. T. Blackstock, eds.), pp. 233–261, San Diego: Academic press, 1998.

[60] Y. Jing, M. Tao, and G. T. Clement, "Evaluation of a wave-vector-frequency-domain method for nonlinear wave propagation," *The Journal of the Acoustical Society of America*, vol. 129, no. 1, pp. 32–46, 2011.

[61] J. W. Goodman, *Introduction to Fourier Optics*. Englewood, CO: Roberts & Company Publishers, 3 ed., 2005.

[62] OpenMP Architecture Review Board, "The OpenMP® API specification for parallel programming." http://openmp.org/wp/. Accessed: 2014-04-15.

[63] "Use and create mex-files," in *The MathWorks Documentation Center*, The MathWorks Inc., 2013. http://www.mathworks.se/help/matlab/. Accessed: 2013-03-04.

[64] S. Gorlin, "Why is Matlab so slow?," *Part of MIT lecture: "Advanced Matlab"*, 2008. http://www.scottgorlin.com/wp-content/uploads/2008/01/day2.pdf. Accessed: 2013-03-04.

[65] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture.," *Intel Technology Journal*, vol. 6, no. 1, 2002.

[66] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su, "Intel® OpenMP® C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance.," *Intel Technology Journal*, vol. 6, no. 1, 2002.

[67] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, vol. 10. The MIT Press, 2008.

[68] "Matlab data," in *The MathWorks Documentation Center*, The MathWorks Inc., 2014. http://www.mathworks.se/help/matlab/. Accessed: 2014-03-04.

[69] J. Kowalik and T. Puźniakowski, *Using OpenCL: Programming Massively Parallel Computers*, vol. 21. IOS Press, 2012.

[70] D. M. Burton, *Elementary number theory*. Tata McGraw-Hill Companies, Inc., 7 ed., 2011.

[71] R. E. Apfel and C. K. Holland, "Gauging the likelihood of cavitation from short-pulse, low-duty cycle diagnostic

ultrasound," *Ultrasound in medicine & biology*, vol. 17, no. 2, pp. 179–185, 1991.

[72] H. Flynn, "Generation of transient cavities in liquids by microsecond pulses of ultrasound," *The Journal of the Acoustical Society of America*, vol. 72, no. 6, pp. 1926–1932, 1982.

[73] J. G. Abbott, "Rationale and derivation of MI and TI —- a review," *Ultrasound in Medicine & Biology*, vol. 25, no. 3, pp. 431–441, 1999.

[74] MATLAB®, *Version 7.13.0.564 (R2011b)*. The MathWorks Inc., Natrick, MA, August 2011. Software.

[75] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms and Applications*. Upper Saddle River, NJ: Pearson Prentice-Hall, 4 ed., 2007.

[76] R. E. Walpole, H. R. Myers, L. S. Myers, and K. Ye, *Probability & statistics for engineers & scientists, Person Education*. Upper Saddle River, NJ: Inc., 2007. ISBN 0-13-204767-5.

[77] G. Rünger, "Parallel programming models for irregular algorithms," in *Parallel Algorithms and Cluster Computing*, pp. 3–23, Springer, 2006.