Delft University of Technology

Bachelor thesis

Bachelor Applied Physics & Applied Mathematics

Uniform Space and Adaptive Time Algorithms for solving the Westerveld Equation in Julia.

Author
Siem de Wit

Supervisors
Dr. Domenico J.P. Lahaye
Dr. ir. Martin D. Verweij

Delft University of Technology,

Faculty of Applied Sciences &
Faculty of Electrical Engineering, Mathematics and Computer Science,

Section of Acoustical Wavefield Imaging &
Department of Numerical Analysis July 8, 2022

TU Delft Delft University of Technology

Summary

In this Bachelor project we researched the applicability of finite difference methods to solve the multi dimensional Westervelt equation. The Westervelt is an equation describing the propagation of a non-linear wave. The goal of this project was to solve the full Westervelt equation in higher dimensions. We used Julia for our implementations of the finite difference method. The choice of Julia allowed us to easily write high performing code.

We were unsuccessful in modelling the full Westervelt equation since we were unable to implement attenuation into our models. We successfully implemented the non-linear wave equation without attenuation for a number of models, including a three dimensional model. The inability to implement attenuation into our models hampered the flexibility of our models. When the non-linearity created a shock front, the top of the wave would experience numerical errors which quickly made the whole solution unphysical and ultimately unstable. An added implementation of attenuation would dampen the leading edge of the wave preventing it from becoming too steep. Additionally we made two comparisons with the linear wave equation to verify the correctness of our implementations.

We used a uniform grid to discretize the spatial dimensions. After which we used non-stiff FDM solvers with adaptive time stepping algorithms to solve for the remaining time dimension. We analyzed the use of stiff FDM solvers but these proved too computationally expensive to be viable, especially for our 3D model with 64.000 points.

Contents

1 Introduction

Medical ultrasound techniques are a family of imaging techniques making use of ultrasound. Ultrasound means sound waves which have frequencies higher then the upper limit of human hearing, which is approximately 20 kilohertz. Apart from the high frequency ultrasound waves are simply regular sound waves. Ultrasound waves see common usage in imaging techniques because their higher frequency allows for a greater resolution. In modern medical ultrasound imaging the amplitude of the generated sound waves exceeds 1 MPa. When using such a high amplitude certain side effects can no longer be ignored. These include the compression and expansion of tissue which change the density and thus the wave speed in the tissue. More specifically an increase in density causes an accompanying increase of the wave speed. This causes the peaks and valleys of a pressure wave to propagate at different speeds, the top of the wave will move faster than the lower portions of the wave and thus overtake it. This will cause the formation of a shock front where the pressure wave suddenly spikes. Taking this effect into account we get a new wave equation, the so-called Westervelt equation

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{dt^2} + \frac{\delta}{c^4}\frac{\partial^3 p}{\partial t^3} = -\frac{\beta}{\rho_0 c^4}\frac{\partial^2 p^2}{\partial t^2}$$

With p the pressure field, c the sound speed, ρ_0 the ambient density, δ the attenuation coefficient and β the coefficient of non-linearity. Here β is a dimensionless constant depending on the material which the wave is traveling trough. Since it precedes the derivative of the pressure wave squared this is called the non-linearity coefficient. A list of different non-linearity coefficients for different materials is given in [12]. For this equation there are no known analytical solutions, thus we will solve it with the use of numerical methods. For this we have chosen the Finite Difference Method(FDM) using an uniform grid space discretization. We will implement our numerical methods in a fairly new programming language called Julia.

Structure of this thesis

In chapter 2 we will first give an overview of the theory relevant to our research and expand some more on the Westervelt equation. Then in chapter 3 we will define our model, this includes first a physical explanation of the model and secondly we will give a precise mathematical formulation. In chapter 4 we will give a theoretical explanation of finite difference methods, as well as the spatial discretization used. In chapter 5 we will give some information on the Julia programming language as well as explaining our reasoning for choosing this language instead of more established languages. In chapter 6 we will show the results of our simulation, along with a comparison to the linear wave equation in 1D. Finally in chapter 7 the conclustion and discussion can be found.

2 Theory

2.1 Linear Wave Equation

In 1746 d'Alembert discovered the one dimensional wave equation[4]. The linear or standard wave equation is a commonly known second-order partial differential equation which describes the propagation of all sorts of waves. In our case we look at sound waves which are in effect pressure waves. The 1D wave equation stated here

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

Where c is the speed of sound. Because the standard wave equation is linear it is easily solvable with analytical methods in a variety of ways. D'Alambert also provided a analytical solution for the 1D wave equation which is now known as d'Alambert's formula shown. We will look at a simplified case where the the wave has no initial velocity but only initial displacement, this gives us the following expression for the solution

$$u(x,t) = \frac{1}{2}[u(x-ct,0) + u(x+ct,0)]$$

The solution has a left travelling part along with a right travelling part. In general for a certain initial displacement we will see half of the initial displacement travel left and the other half travel right. For the derivation of the linear wave equation a number of assumptions have to be made. One of these is that the wave speed is independent of the amplitude of the wave. For most cases this is a perfectly fine assumption, however for medical ultrasound imaging with large amplitudes and high frequencies this assumption can no longer be made. This leads us to search for a better equation describing high energy high frequency waves.

2.2 Non-Linearity

A sound wave travelling trough a material is described as a change of pressure propagating trough the material. This change in pressure also affects the speed of sound in the material. Since the change in pressure varies for different parts of the wave also the speed of sound will vary. In our case this causes the top of the wave, where the pressure is highest, to travel faster than parts of the wave where the pressure is lower. This will lead to the higher parts of the wave overtaking the lower parts. Since the higher parts of the wave cant actually overtake the lower parts, this will cause a shock wave to be formed where the pressure wave abruptly changes. An additional effect is the attenuation which causes the wave to lower in intensity as it travels trough the material, this is caused by the wave losing energy to the environment by compressing the material and subsequently heating it up. These two effects together are demonstrated more clearly by the below figure

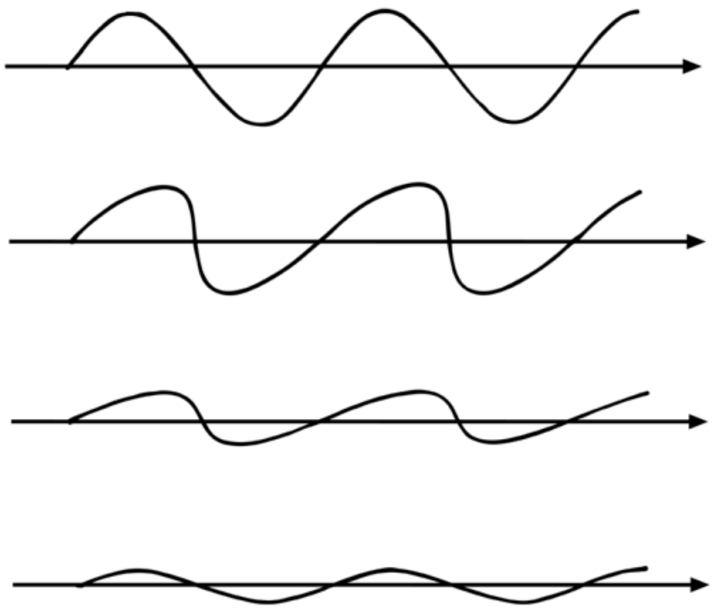


Figure 1: Example of how a wave described by the Westervelt equation can propagate[19].

In the example above the formation of a shock front can be seen with the wave becoming more steep. However before the wave can form a full shock front the effects of attenuation cause the forming shock front to become less severe. Then in the 3rd and 4th plots the effect of attenuation can again be seen by causing the wave to decrease in intensity.

2.3 Westervelt Equation

The Westervelt equation takes into account the effects of non-linearity as well as attenuation[8]. It is fully stated here

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{dt^2} + \frac{\delta}{c^4}\frac{\partial^3 p}{\partial t^3} = -\frac{\beta}{\rho_0 c^4}\frac{\partial^2 p^2}{\partial t^2} \tag{1}$$

With $\nabla^2 = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2}$ the laplacian. Where ρ_0 is the ambient density, c is the sound speed, δ is the sound diffusivity and β is the non-linearity coefficient. If we set $\beta = \delta = 0$ we recover the linear wave equation as expected. The goal of this project was to model the full Westervelt equation in higher dimensions using finite difference methods. In this we were unsuccessful in implementing attenuation into our model. In this report we will thus look at the Westervelt equation with δ equal to zero. The reason for this is discussed in more detail in the conclusion in chapter 7. This leaves us with the modified Westervelt equation without diffusivity:

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = -\frac{\beta}{\rho_0 c^4}\frac{\partial^2 p^2}{\partial t^2} \tag{2}$$

This is the equation we will focus on in this report. We will often refer to this as the "non-linear wave equation".

3 Model

In this section we will give an overview for the different models we analyzed. We have prioritized the 2D driven model as our main model. For this model we will give a physical explanation as well as a mathematical formulation. For the simpler auxiliary models we will give a mathematical formulation along with a brief physical model statement.

3.1 Physical Model

We will start of with our main model, which is that of a Driven 2D rectangular domain. We have chosen for a rectangular domain because of the difficulty of implementing an irregular domain with the use of FDM. We have chosen to have a driven side of the domain in contrast to a model only using non-zero initial conditions. The reason for this is because it more closely resembles a real world scenario. In medical acoustic imaging a sound wave gets created by a transducer, the waves then enter the subject of the examination. Then the waves then reflect of of changes in the density of the subject, after which a receiver can measure these reflected waves and thus make an image of the subject.

Sound waves which have passed the subject should not further interfere in our system. Thus we want appropriate boundaries in our model which do not reflect the incoming waves. This leads to so-called radiating boundary conditions. The model is visualized in the figure below

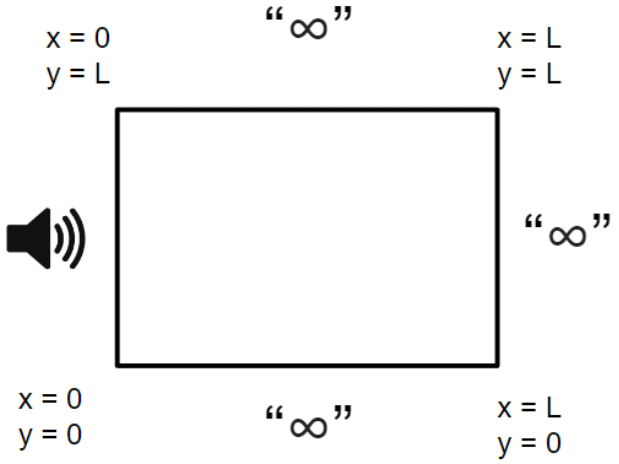


Figure 2: A visualization of the 2D model we will be studying.

In the model the transducer is shown as a speaker which will transmit waves into our domain. We have chosen for a square domain because of the more simple implementation. At the x=L,y=0,L boundaries the radiating boundary conditions are shown as an infinity symbol, as if the waves beyond our domain would go to infinity.

For the wave equation we have chosen to model the non-linear wave equation as specified in equation 2. This model does not include the attenuation as specified in the full Westervelt equation. For all models unless specified differently we will use the non-linear wave equation without attenuation.

3.2 Mathematical Formulation

In this section we will give a mathematical formulation of the different models used in the same order in which they will appear in the Results in chapter 6. First we will look at two different one dimensional models. The reason we have chosen to look at these models is to show our implementations correspond to known solutions for the linear wave equation and to make a comparison between linear and non-linear waves.

3.2.1 Initial Conditions Linear Wave 1D

The first model is used to make a comparison with the analytical solution for the one dimensional linear wave equation. The domain used is $x \in [0, L],\ t > 0$

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2 \nabla^2 u(x,t)$$

$$u(x = 0, t) = 0$$

$$u(x = L, t) = 0$$

$$u(x, t = 0) = f(x)$$

3.2.2 Driven Non-Linear Wave 1D

The second model is used to make a comparison between the computed solutions for the one dimensional linear wave equation and the one dimensional non-linear wave equation. We will list both equations along with the shared initial and boundary conditions. The domain used is $x \in [0, L,\ t > 0]$

$$\frac{\partial^2 u_1(x,t)}{\partial t^2} = c^2 \nabla^2 u_1(x,t)$$

$$\frac{\partial^2 u_2(x,t)}{\partial t^2} = c^2 \nabla^2 u_2(x,t) + \frac{\beta}{\rho_0 c^2} \frac{\partial^2 u_2(x,t)^2}{\partial t^2}$$

With both equations having the same initial and boundary conditions

$$u(x = 0, t) = f(t)$$

$$u(x = L, t) = 0$$

$$u(x, t = 0) = 0$$

3.2.3 Driven Non-Linear Wave 2D

Now we will look at our main model, the 2D driven non-linear wave equation with radiating boundary conditions. For the radiating boundary conditions we used conditions as described by Broeze (1992)[10]. We use the non-linear wave equation without attenuation, resulting in the following problem statement

$$\frac{\partial^2 u(x,y,t)}{\partial t^2} = c^2 \nabla^2 u(x,y,t) + \frac{\beta}{\rho_0 c^2} \frac{\partial^2 u(x,y,t)^2}{\partial t^2}$$

$$u(x, y, t = 0) = 0$$

$$u(x = 0, y, t) = f(y, t)$$

$$\frac{\partial u(x = L, y, t)}{\partial t} + \frac{\partial u(x = L, y, t)}{\partial x} = 0$$

$$\frac{\partial u(x, y = 0, t)}{\partial t} - \frac{\partial u(x, y = 0, t)}{\partial y} = 0$$

$$\frac{\partial u(x, y = L, t)}{\partial t} + \frac{\partial u(x, y = L, t)}{\partial y} = 0$$

3.2.4 Initial Conditions Non-Linear Wave 3D

Finally we will look at a simpler model in three dimensions. We will use a model with non-zero initial conditions and homogeneous Dirichlet boundary conditions at every boundary. We use a domain A of size LxLxL. We use the non-linear wave equation without attenuation, resulting in the following problem statement

$$\frac{\partial^2 u(x,y,z,t)}{\partial t^2} = c^2 \nabla^2 u(x,y,z,t) + \frac{\beta}{\rho_0 c^2} \frac{\partial^2 u(x,y,z,t)^2}{\partial t^2}$$

$$u(x, y, z, t = 0) = f(x, y, z)$$

$$u(x, y, z, t) = 0 \qquad on \qquad \partial A$$

4 Finite Difference Methods

In this project we have chosen to research the use of finite difference methods(FDM) to solve the Westervelt equation. Part of the reason we have chosen to research the use of FDM as a method to solve the Westervelt equation is because it is a method that is not commonly used to solve the Westervelt equation. Thus there is still a great deal of progress to be made about the application of FDM to the Westervelt equation. Another reason is because previous students have examined primarily the use of finite element methods(FEM) to solve the Westervelt equation. FEM methods in general handle irregular boundary conditions better but are more difficult to implement. The choice of FDM allowed us to develop a more advanced model in less time than would have been needed to implement it with FEM.

Finally the FDM is particularly well-suited for solving our stated problem formulation. In our model as stated in chapter 3 we are using a rectangular domain which is easy to implement, the choice of a rectangular domain itself is understandably influenced by the fact that we use FDM. Added is the easiness of implementing different algorithms to solve for the time dimension.

4.1 Finite Differences

If we look at a function u(x) defined on the closed interval [0,L].

$$u(x),\ x \in [0, L]$$

We can approximate the derivative of u(x) by first looking at the definition of the derivative.

$$\frac{\partial u}{\partial x} = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h}$$

By then simply taking h to be small we retrieve the forward difference formula for the first derivative. Since h is small and not approaching zero there will be an accompanied error. In this case the error is proportional to h. Thus reducing h also leads to a smaller error.

$$\frac{\partial u}{\partial x} = \frac{u(x+h) - u(x)}{h} + \mathcal{O}(h)$$

In this specific discretization the error is proportional to h. There exist other discretizations which have a different order of accuracy. Lets take for example the centered difference of the first derivative.

$$\frac{\partial u}{\partial x} = \frac{u(x+\frac{h}{2}) - u(x-\frac{h}{2})}{h} + \mathcal{O}(h^2)$$

For this finite difference the error is proportional to h^2, we then say this finite difference has an order of 2. In general for any order of derivative a finite difference formula can be found with a desired order of accuracy (Fornberg 1988)[7]. For example, Fornberg shows an 6th order central difference for the first derivative:

$$\frac{\partial u}{\partial x} = \frac{(-u(x-3h) + 9(x-2h) - 45u(x-h) + 45u(x+h) - 9u(x+2h) + u(x+3h)}{60h} + \mathcal{O}(h^6)$$

Here the error vanishes very quickly as we lower h. This is exactly what we want, however there is a downside. To calculate the derivative using this 6th order central difference we need to evaluate our function u 6 times. Whereas we only had to evaluate u twice to get a 2nd order approximation. In the next subsections it will be clear how the choice of finite difference effects the computational power required to solve our differential equation. We will also expand on our choice of finite difference formula.

4.2 Discretization in space

We will use the so called method of lines to discretize the Westervelt equation in all space dimensions, leaving only the time dimension continuous. This will leave us with a system of coupled ODE's which are then integrated in time by the use of FDM. For this we will use a 2nd order central difference formula for the laplacian. We will go trough the whole process of discretizing the Westervelt equation in 1D here. A full derivation for the two and three dimensional Westervelt equation can be found in the appendix.

$$\nabla^2 u(x,t) = \frac{\partial^2 u(x,t)}{\partial t^2} \approx \frac{u(x-\Delta x,t) - 2u(x,t) + u(x+\Delta x,t)}{\Delta x^2}$$

We can then discretize the dependence of u on x. Namely we make a grid of N intervals and correspondingly N+1 nodes. This gives $\Delta x = \frac{L}{N}$. We can then define $u_n(t) = u(n\Delta x, t)$. This lets us write the above equation as

$$\nabla^2 u_n(t) = \frac{u_{n-1}(t) - 2u_n(t) + u_{n+1}(t)}{\Delta x^2} \quad \forall n \in [1, N]$$

The above equation is only defined for the interior points of the domain. If n where to be 0 or N+1 then we would need to evaluate u at points where it is not defined. Later we will see how to use the boundary conditions to formulate an equation for the boundary points. The discretization of the laplacian will now allow us to rewrite the Westervelt equation for our discretized u. Since we have not yet defined the laplacian for points on the boundary we will first only look at the interior points. This gives us the following discretized Westervelt equation

$$\frac{u_{n-1}(t) - 2u_n(t) + u_{n+1}(t)}{\Delta x^2} - \frac{1}{c^2}\frac{\partial^2 u_n(t)}{\partial t^2} = \frac{\beta}{\rho_0 c^4}\frac{\partial^2 u_n(t)^2}{\partial t^2} \quad \forall n \in [1, N]$$

We can then use the following expansion which is just a rewriting of derivatives

$$\frac{\partial^2 (u_n(t))^2}{\partial t^2} = 2u_n(t)\frac{\partial^2 u_n(t)}{\partial t^2} + 2\left(\frac{\partial u_n(t)}{\partial t}\right)^2$$

Finally we can rewrite the discretized Westervelt equation to isolate the $\frac{\partial^2 u_n(t)}{\partial t^2}$ term, while simultaneously using the above simplification to retrieve

$$\frac{\partial^2 u_n(t)}{\partial t^2} = c^2\frac{u_{n-1}(t) - 2u_n(t) + u_{n+1}(t)}{\Delta x^2} - \frac{2\beta}{\rho_0 c^2}\left(u_n(t)\frac{\partial^2 u_n(t)}{\partial t^2} + \left(\frac{\partial u_n(t)}{\partial t}\right)^2\right) \tag{3}$$

4.3 Solving in time

Finite difference methods are a class of numerical techniques which utilize the finite differences as demonstrated above to solve ordinary differential equations. Since we have already discretized our partial differential equation into a system of coupled ODE's, we can now utilize a FDM to solve the remaining time dependency. In practice we will use FDM algorithms which are more complicated.

To demonstrate how a FDM works we will run trough the solving steps for a simpler example. Lets analyze the following equation

$$\begin{aligned} \frac{\partial u(t)}{\partial t} &= 1.01 * u(t) \quad t > 0 \\ u(t=0) &= 0.5 \end{aligned} \tag{4}$$

This simple ODE has the analytical solution

$$u(t) = 0.5e^{1.01t}$$

We will now use a first order forward difference applied to the time derivative, namely

$$\frac{\partial u(t)}{\partial t} \approx \frac{u(t + \Delta t) - u(t)}{\Delta t}$$

Here again the discretization introduces a numerical error of order $\mathcal{O}(\Delta t)$. Let's discretize u with the following definition

$$u_n = u(n\Delta t); \quad u_0 = 0.5$$

Let us now set substitute our discretization into the ODE

$$\frac{u_{n+1} - u_n}{\Delta t} = 1.01 * u_n$$

We can reorder this equation to isolate u_{n+1}

$$u_{n+1} = u_n + \Delta t \cdot 1.01 \cdot u_n$$

This is called a finite-difference equation. Since we know u at t=0 we also know u_0, this will now allow us to repeatedly use the finite-difference equation to get u for higher values of n. The method described above is also called the Euler forward method since it uses the forward finite difference. While discretizing the first order derivative we introduced a numerical error. This error is proportional to Δt. So we have to remember to keep Δt small and ideally as small as possible. Suppose however we want to solve the differential equation above up to a time T, if we then lower Δt we would need to take more steps to arrive at the final solution. In FDM there is always this consideration between taking smaller steps to induce a smaller error, and not taking too small steps which would make the solve take too much time.

As an example we have implemented the above described method in Julia to help us gain a better understanding of this method. In figure 3 the results are shown for two different values of Δt along with the true solution.

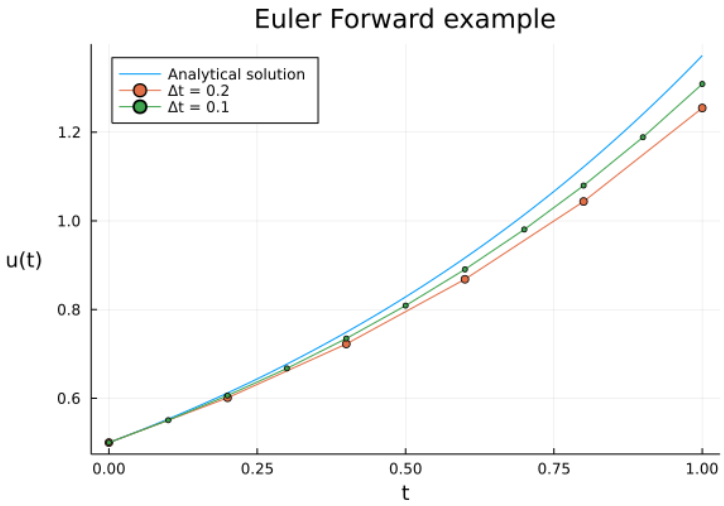


Figure 3: FDM solves of ODE (4) for different Δt compared to the true analytically derived solution

The Euler forward method is one of the most basic implementations of a FDM. A downside of the simplicity of this method is its numerical error, which scales linearly with Δt. This is not the main problem however, a numerical error is always expected and at least for the above ODE it returns a sensible result. For certain combinations of equations and stepsizes however the Euler forward method fails to return a numerically stable result. In the subject of numerical differential equations there exist various concepts of stability. A commonly used procedure for stability analysis is the so called "Von Neumann stability analysis" first given a rigorous definition in an article by Von Neumann [3]. For now however the exact definition of stability/instability is not important, we can broadly regard instability as the failure of a FDM to come to a solution resembling the true solution. In the above example we can see the solutions acquired by the Euler forward method are not exact but do resemble the true solution, we can also see the calculated solution becomes closer to the true solution when we lower the step size .

We will now look at a different ODE for which the Euler forward method partially fails to return a stable solution. The following problem is similar to the previous example but has a slight but meaningful alteration.

$$\begin{aligned} \frac{\partial u(t)}{\partial t} &= -2u(t) \quad t > 0 \\ u(t=0) &= 0.5 \end{aligned} \tag{5}$$

Solving this ODE is trivial which gives us

$$u(t) = 0.5e^{-2t}$$

We can again implement the same method as above for again different values of Δt. In figure 4 we can see the solution plotted for 2 varying values of Δt along with the analytically calculated true solution.

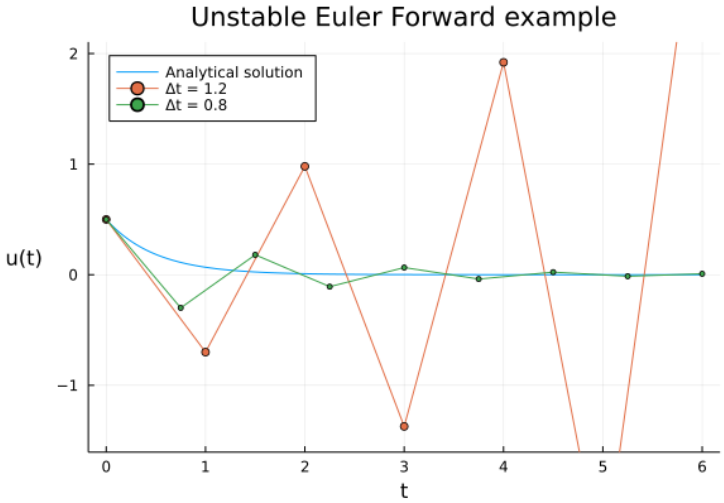


Figure 4: FDM solves of ODE (5) for different Δt compared to the true analytically derived solution

For this slightly different ODE we see radically different behavior for different step sizes. We will first look at the solution $\Delta t = 0.8$. The solution is not great, it severely deviates from the true analytically derived solution. For larger t it seems the method approaches the true solution more closely, as the absolute error error is indeed approaching zero but this is misleading since the true solution also rapidly approaches zero. In this case it is more useful to analyze the relative error, which in this case is our derived solution divided by the true solution. In contrast to the absolute error which is mostly decreasing, the relative error only increases. At time $t = 8$ the derived solution is off by a factor of 1000x.

Now looking at the behaviour for $\Delta t = 0.8$ we can see the derived solution does not approach the true solution in the absolute sense. In fact it oscillates around $u = 0$ with the oscillations only growing larger. This is unstable behaviour. Luckily with FDM you can always simply decrease the step size. In this example it is clear the Euler forward method can be stable with small enough step sizes. In practice certain problems can be found for which the method has to take impractically small step sizes to come to a stable solution.

The possibility of instability along with the slow convergence of the error lead us to search for higher order methods. In the study of FDM most solving algorithms used are Runge-Kutta methods.

4.4 Runge-Kutta methods

Runge-Kutta(RK) methods are a family of explicit and implicit methods, they were developed by Carl Runge and Martin Kutta around the year 1900. Runge in his paper of 1895[16] first explored different methods to solve an ODE of the form

$$\frac{\partial y(t)}{\partial t} = f(t, y(t)), \quad y(t_0) = y_0 \tag{6}$$

Then in 1901 Kutta took the analysis of the later called Runge-Kutta methods further[11], and introduced among others a method of order 5. Kutta also introduced a classification of methods of order 4,and introduced the now well known Runge-Kutta 4th order method(RK4).

For a expansive description of Runge-Kutta methods i refer to the textbook "An Introduction to Numerical Analysis" by Atkinson(1978)[1]. Most RK methods used are mixed implicit and explicit. As an example we will outline the RK4 method applied to equation (6) stated above. Defining a step-size $\Delta t > 0$ and discretization $y_n = y(n\Delta t)$. We can now state the formulation of RK4.

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t$$

$$t_{n+1} = t_n + \Delta t$$

With $k_{1,2,3,4}$ defined in the following way

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{\Delta t}{2}, y_n + \Delta t \frac{k_1}{2})$$

$$k_3 = f(t_n + \frac{\Delta t}{2}, y_n + \Delta t \frac{k_2}{2})$$

$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3)$$

Here y_{n+1} is the value given by RK4 for y at the next time step. To implement this we only need to know y at time $t = n\Delta t$. In contrast we need to know f at times after the current time step. Specifically f is taken as the weighted average of multiple evaluations of f. This comes with the downside of having to evaluate f four times instead of only once in the case of Euler forward, which is especially costly if f is very difficult to compute.

The upside of taking an average of multiple evaluations is an increase in the order of the truncation error. The RK4 method as hinted by the 4 in its name is a fourth-order method. Meaning that the global truncation error is on the order of $\mathcal{O}(\Delta t^4)$. This will allow us to reach a lower error far more quickly than with the use of Euler forward.

The RK4 method described is still an explicit method. In general explicit Runge-Kutta methods are not suitable for computation of stiff differential equations. To produce stable results often the step size has to be taken unreasonably small. Implicit Runge-Kutta methods are in general far more suited for the solving of stiff equations. Implicit methods are in general more difficult to compute, because the method is implicit there does not exist a explicit formula to compute the function at the next time step. Implicit methods need to solve a system of equations at every time step. This greatly increases the computational cost. The reason they are superior for stiff equations comes from the fact that explicit methods will often have to take unreasonably small steps to avoid instability. Altough sometimes an explicit method can outperform a implicit method while taking steps far smaller, simply because it is faster to calculate. Often an implicit method can then take larger steps while still producing stable results.

The simplest example of an implicit method is the backward Euler method. Lets look at the following ODE

$$\frac{\partial y(t)}{\partial t} = -y^2$$

Which using the backward finite difference gives

$$\frac{y_{n+1} - y_n}{\Delta t} = -y_{n+1}^2$$

Which can be written as

$$n + 1 + \Delta t y_{n+1}^2 = y_k$$

We know y_k which means we can find an expression for the value of y at the next time step. To do this we need to isolate y_k, which in this case requires solving a quadratic equation. Which finally gives us

$$y_{n+1} = \frac{-1 + \sqrt{1 + 4\Delta t y_n}}{2\Delta t}$$

In this case it was quite simple to solve for y_{n+1}, in practice however the equation to be solved is much more difficult than a simple quadratic. Often no analytical solution exists. Then we would need to use root-finding algorithms to find y_{n+1}. In our case the non-linearity of the Westervelt equation makes the root-finding problem more difficult.

Additionally a technique which finite difference methods can use is adaptive time stepping. All methods yet shown use fixed time-stepping. With adaptive time stepping the user can define a desired threshold for the global truncation error. The algorithm will then keep track of the local truncation error. If the error rises too high the algorithm can adaptively lower the step size, conversely if the local error would result in a global error smaller than the user specified then the algorithm can increase the step size to increase performance. Much research time is invested on finding optimal adaptive algorithms. For example the Tsit5[17] method often used in DifferentialEquations.jl features a fourth-order, five-stage explicit Runge-Kutta method with the embedded error estimator of Tsitouras. Efficiently calculating the error can save a great amount of computation time.

Finally finite difference methods can make use of a technique called method switching. In this case the algorithm starts with implementing an explicit method which generally allows it to be quicker than implicit methods. Then when the algorithm encounters stiffness it can automatically switch to using a completely different implicit method. These so-called Auto-Switching Algorithms are especially useful when the stiffness of an equation is unknown. This introduces minimal overhead for a non-stiff equation while still allowing the algorithm to solve stiff equations. This allows us to use the best of both worlds regarding implicit and explicit methods.

5 Julia

Julia is a very new programming language which first appeared in 2012, and is rapidly gaining popularity around 2019. Most readers of this report are probably less familiar with Julia than other languages frequently used for numerical programming, other languages which are also vastly more established in the field of numerical programming. As such i feel the need to thoroughly explain the choice for Julia. And use this opportunity to inform the reader about the great language that is Julia and hopefully have them consider Julia for their next project.

5.1 Julia

Julia[2] is a high-level, high-performance, dynamic programming language. Although it is a general-purpose language, it is especially suited for numerical programming.

5.1.1 Technical Details

For a full technical review i refer to a paper by the founders of Julia from 2017[2]. I will however give a brief review of how Julia works. Julia is an open source dynamically typed language which with multiple dispatch as its core paradigm. The syntax is similar to python. Julia uses a JIT(Just-in-time) compiler that is sometimes referred to within the Julia community as "just-ahead-of-time"(JAOT) because Julia by default compiles all code to machine code before executing. Julia is garbage-collected, which allows the user to not have to worry too much about memory usage, especially there is virtually no chance of a memory leak occurring. For better performance however some thought is needed regarding memory usage. Julia is especially suited for linear algebra featuring level 3 BLAS support. Julia uses a built-in package manager with which you can install packages without leaving your Julia environment. A goal of Julia is to enable easy calling of other languages within Julia code, this allows for easy reusing of existing codebases. Fortran and C code can be called natively within Julia, while other languages such as R,Python,Java,C++ and Matlab are able to be called with the use of specific packages. Julia can also be embedded in C,Python and more. Julia also features advanced Unicode support, which can help greatly improve the readability of code. By using mathematical symbols which can be help make the code more legible for the coming from a mathematical background

5.1.2 Two language problem

Julia aims to solve the so called two language problem, resolving this problem is the core guiding ideology of Julia. A popular phrase in the Julia community is “Walk Like Python; Run Like C.”, which highlights this goal aptly. The two language problem is a constant trade-off that developers have to make when choosing a programming language to use for their project. When they chose a language they have to generally choose between languages that are easier for humans to write, or faster for computers to run. For example Python is a language that is generally considered to be very simple to write, the downside of this simplicity is its worse performance. In contrast C is generally considered to be more difficult to write yet very fast. A common workaround for the two language problem is to write part of the code in a high-level language such as R and Python. And then rewrite the performance critical parts ie, the parts which take the most time to execute, in a lower-level language like C or C++. This will result in code which will still execute fast because the parts of the code which are most computationally demanding are written in fast languages. The downside of this dynamic is often having to write code in a lower-level language which has already been written in a higher level language. This is hugely inefficient and can lead to errors. In a 2018 article with TechRepublic[9], Prof. Alan Edelman a co-founder of Julia talks of organizations prototyping code in a higher-level language and then having to "hire a team of programmers to recode it in a fast language". This highlights exactly the two language problem.

Another downside of the two language problem can be shown best by looking at the use of Numpy in Python. Thanks to Numpy it is still possible to do fast linear algebra in Python. But the moment you have to use native Python code to implement something it is orders of magnitude slower. In Julia however simple for loops can often be faster than vectorized versions of the same function.

Julia aims to be easy to write and simultaneously have very fast execution times, which would allow organizations to prototype and deploy in the same language, thus resolving part of the two language problem. And Julia is indeed very fast , in 2021 a NASA modelling group switched from MATLAB to Julia and saw a corresponding 15000x increase in performance[5], in all fairness this is an extreme example and 10x to 100x increases in performance compared to Python/Matlab are more common.

Another reason why we have chosen Julia is because of the ease of use of the interactive environment. Julia supports Jupyter notebook style programming. For me this is a very intuitive way of programming, which allows for easy prototyping without the need of manual compilation. Our choice for Julia is also largely influenced by the SciML(Open Source Scientific Machine Learning) environment,this is a suite of packages designed for the use in both numerical simulation methods and methodologies in scientific machine learning. In particular

the package DifferentialEquations.jl was of great interest to us. The aim was to use the better performance of the Julia language combined with the powerful SciML environment, compared to previous students' choice of Matlab, to allow us to achieve a more advanced model. We have decided not to fully implement the FDM ourselves but instead use a package which includes a large variety of FDM solvers. The reason for this is because simple implementations like Euler forward scale badly when you want a lower error. Aditionally it can be hard to implement the more advanced solvers manually. Aditionally the package we have chosen allows for a number of features which can speed up calculations.

5.2 DifferentialEquations.jl

DifferentialEquations.jl[14] is a package within the SciML environment developed for high-performance solving of various types of differential equations. Some types of equations which are able to be solved include ODEs, SODEs, DAEs and many more, a full overview can be found at https://diffeq.sciml.ai/stable/. Additionally DiffEq.jl supports GPU acceleration,arbitrary precision and arbitrary array types. In principle any custom type which has a defined adding operator can be used. More specifically this allows support for array/matrix types, which we will use.

An expansive list of FDM solvers is available for the use with ODEs. Most of these solvers are Runge-Kutta methods, which are internally divided into solvers more suited for Stiff or Non-Stiff equations. Aditionally a large number of solvers from other languages/packages is available for use within DiffEq.jl, these include solvers from MATLAB/Python/R as well as support for the SUNDIALS C library.

A crucial feature of DiffEq.jl is the automated solver detection, which allows the package to automatically choose an ideal solving algorithm for the specific problem stated[15]. Additionally the user can choose to give certain hints to the package such as whether the equation is stiff or non-stiff. A manual choice of solver can always be specified. Finally the user can specify the desired global error.

We will now show a simple example to highlight the workflow when making use of DiffEq.jl. This is not meant as a comprehensive overview of the package but solely as a example to give the reader an idea of the workflow, as well as an simple example of the syntax used in Julia in general. We again look at the ODE specified in 4 stated here again for readability

$$\begin{aligned} \frac{\partial u}{\partial t} &= f(u,t) = 1.01 \cdot u \\ u(t=0) &= 0.5 \end{aligned} \tag{7}$$

```
f(u,p,t) = 1.01*u
u0 = 0.5
tspan = (0.0,1.0)

prob = ODEProblem(f,u0,tspan)
sol = solve(prob)
plot(sol)
```

Figure 5: All the code needed to solve and subsequently plot equation 4

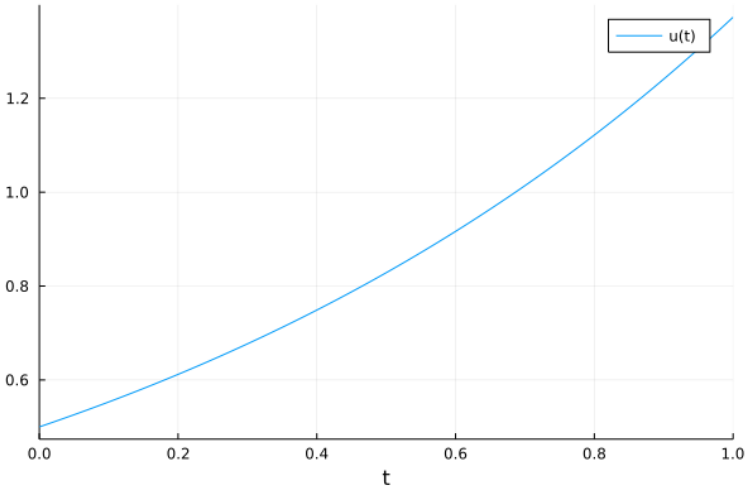


Figure 6: Diffeq.jl solve of ODE (4)

For a performance benchmark of Julia compared to other languages we use benchmarks provided by the SciML organization[13]. In these benchmarks different solvers are compared from different languages an libraries. In particular algorithms from Julia, Matlab, Scipy(Python), deSolve(R) and Sundials(C) are compared for 2 stiff and 2 non-stiff problems. The Julia packages providing support for these libraries provide broad numbers on the overhead of calling these libraries. More specifically, MATLAB and Sundials have neglible overhead, whereas R has a 3x overhead. Scipy is accelerated 3x over Scipy+Numba setups, due to Julia JIT on the ode function providing acceleration compared to Numba. Benchmarks are shown as a plot of time required to solve versus the numerical error to ensure the methods are solving for the same accuracy. Below are shown two of the benchmarks, one stiff and one non-stiff

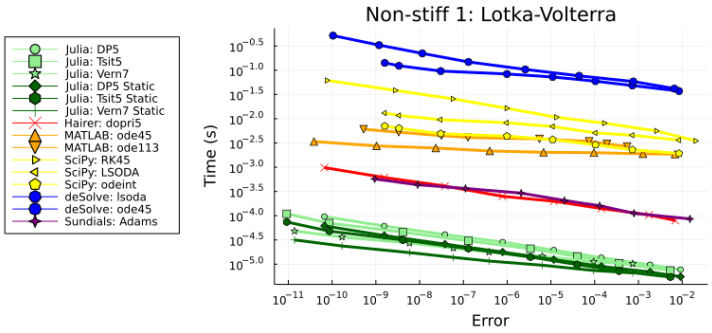


Figure 7: Benchmark comparing different solving algorithms applied to the Lotka-Volterra problem

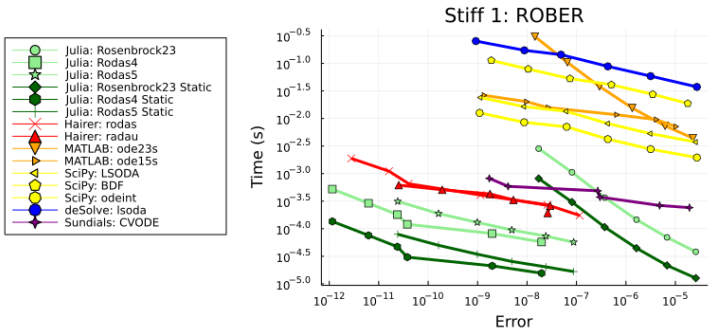


Figure 8: Benchmark comparing different solving algorithms applied to the ROBER problem

From the benchmarks shown above it is clear that DiffEq.jl is a worthy choice of library. Benchmarks like these largely influenced our choice for DiffEq.jl. It should be noted these benchmarks come from the group developing DiffEq.jl themselves, thus it could be valuable to also look at third party benchmarks. For benchmarks of the whole SciML environment including some machine-learning focused comparisons, i refer to (github.com/SciML/SciMLBenchmarks.jl).

6 Results

In this section we will show the results we have achieved for our 2 different models which are formulated in chapter 3. Additionally we will we make an analysis of our implementation of a linear wave in 1D, which we will compare to the known analytical solution. With this we feel we have given enough justification for the physicality of our implementation. Thus we will not make a comparison in higher dimensions for our implementations of the linear wave equation. We have also chosen not to make a comparison with the Burgers' equation, since the Burgers' equation make the assumption of strictly forward propagating waves, which is not the case for our models. For an analysis of the Westervelt equation compared to the Burgers' equation i refer to the work of previous students eg. Zijta 2017[19].

6.1 Implementation

We will know give a very broad high level overview of our implementation. To review the full code for the different models i refer to my GitHub: (github.com/SiempieW/BepWesterveltJulia), the full animations are also shown here. We have solved the problem as a coupled system of two differential equations. This allows us to only involve first derivatives in time.

$$\frac{\partial u}{\partial t} = v$$

$$\frac{\partial v}{\partial t} = c^2 \nabla^2 u + \frac{\beta}{c^2 \rho_0} \frac{\partial^2 u^2}{\partial t^2}$$

- Define u0 as array of 2*N
- Set second halve of u0 as the initial conditions
- Define higher dimensional Laplacian matrix with Kronecker product
- Define f as in equation 7
- Reshape u0 to 1D vector
- Define DiffEq.jl problem
- Let DiffEq.jl solve with specified errors/solver
- Handle solution
- Make pretty plots

We use the physical values provided by Zijta 2017, with δ set to 0. By setting δ to zero we effectively ignore the attenuation term in the Westervelt equation, this counters the original goal of modelling the full Westervelt equation. The reason for this will be discussed in chapter 7.

Table 1: An overview of the parameters use for simulations of the Westervelt equation. Unless stated otherwise these parameters are used

ρ_0	$1000 \frac{kg}{m^3}$	Material ambient density
c_0	$1500 \frac{m}{s}$	Sound speed
β	10	Coefficient of nonlinearity
P_0	10^6 Pa	Maximum pressure amplitude
L	1.5	Domain length
δ	0	Attenuation coefficient
N		Amount of grid points used per spatial dimension
T		Simulation time
ϵ	1e−6	Absolute and Relative error used

6.2 Linear Wave Comparison

To first show our model is physically correct we will make an comparison with a analytically computed 1D Linear wave. First we will show a linear model featuring non-zero initial conditions and homogeneous boundary conditions. Subsequently we will look at a non-linear model where the left hand side is driven and where the initial conditions are zero.

6.2.1 Initial Conditions Linear wave 1D

First the model with non-zero initial conditions. For the initial condition we have chosen a $sin(x)^6$ as a initial peak centered around the midpoint of the domain, this give us a sharp wave which is more suited for comparing to the analytical solution. The desired relative and absolute errors we gave DiffEq.jl were both 1e−6. The solver choice was left undefined, the algorithm then chose a composite solving algorithm consisting of Vern7[18] as a base with stiffness causing switching to Rodas4[6] when required. Constants where as defined in table (1) along with N = 500 and T = 0.5ms, the solve took $\approx 0.15s$ with 26.34k allocations of total size 127.064 MiB. A total of 767 time steps were taken which results in an average Δt of 6.5e−7 This results in the following initial conditions and simulation

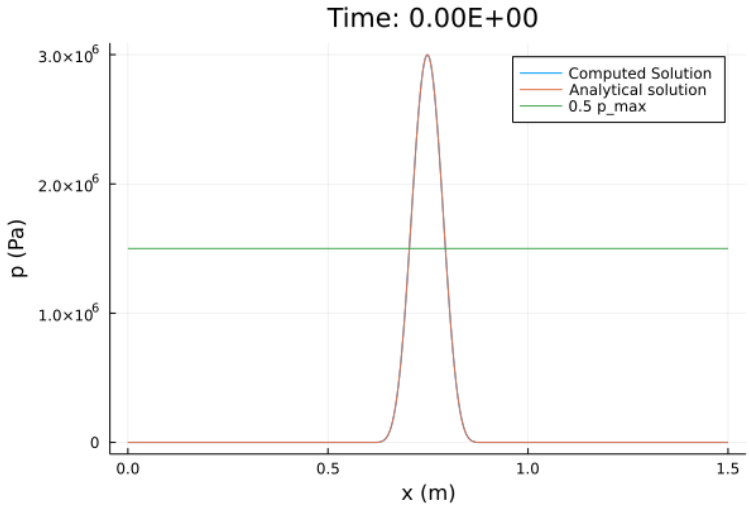


Figure 9: Initial conditions used for comparison with analytical solution for the 1D linear wave equation.

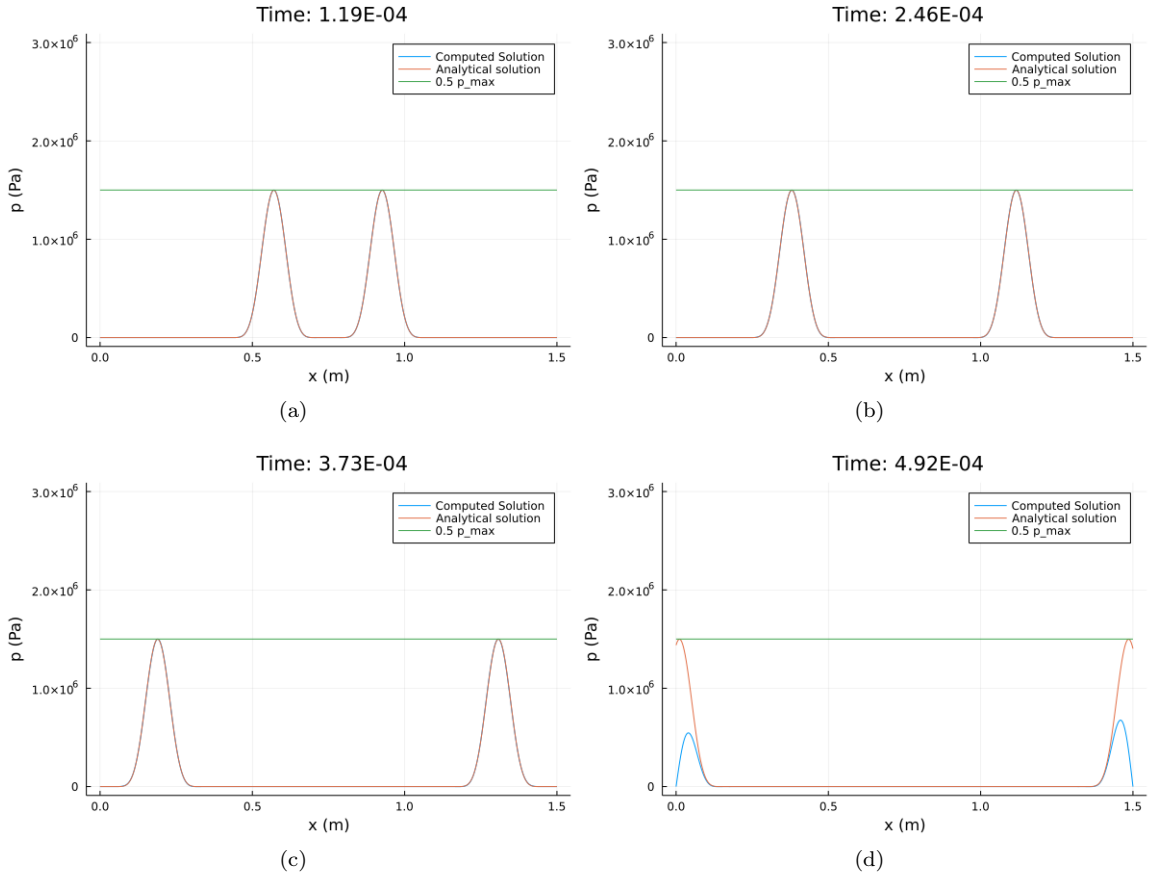


Figure 10: Plot of the FDM solution for the linear wave equation along with the analytical solution for four different time values.

In the above figures it can be seen the computed solution follows the analytical solution closely, once the wave reaches the edge the computed solution differs from the analytical. This difference is because we have not implemented reflection of the boundaries in our analytical solution. A full animation can be found on my GitHub: (github.com/SiempieW/BepWesterveltJulia).

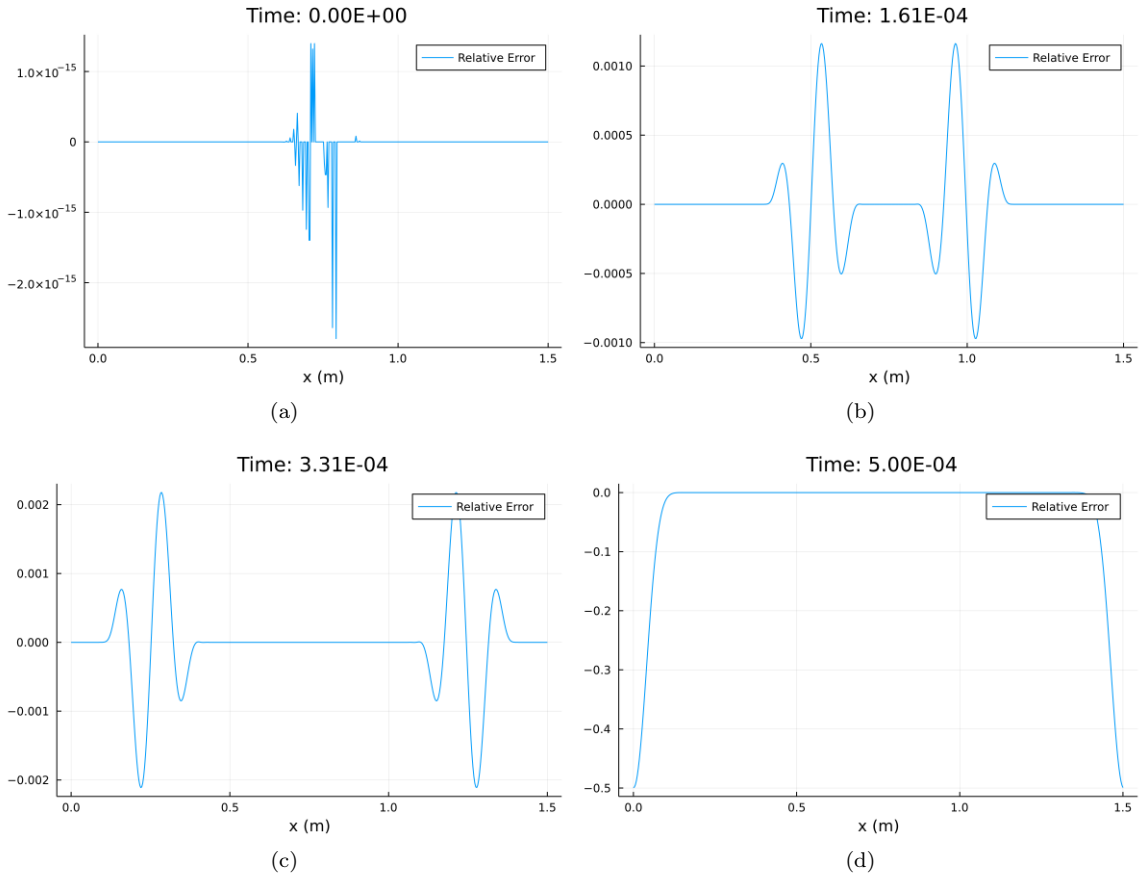


Figure 11: Plot of the relative error of the FDM solution for the linear wave equation for four different time values.

In the above figure the relative error is shown for our computed FDM solution of the linear wave equation. At time $t = 0$ the error is virtually negligible and attributable to floating point errors. For times thereafter however there is a clear error visible. This error likely stems from the spatial discretization used, since the error perfectly develops in time like a wave. An additional reason to support this reasoning is the fact that the error did not change majorly when a finer spatial discretization was used. This leads us to think an error has been made in the implementation of the initial conditions. In the final plot again the error is attributable to the missing implementation of boundary conditions on the analytical solution.

6.2.2 Driven Non-Linear Wave 1D

We will now look at the model with driven left hand side and zero initial conditions. For the driving function we have chosen $sin(f * t)^2$ for $t < 1/5000, f = pi * 5000$. The simulation was run twice, once for the linear wave and once for the non-linear Westervelt equation without attenuation. With again the constants as described in table1, along with N = 500 and T = 1ms. For both simulations the non-stiff Tsit5[17] solver was manually chosen. The linear wave equation was solved in 0.43s with 237K allocations of total size 451 Mib. The linear equation required 4061 time steps with an average $\Delta t = 2.5\text{e}{-7}$. The non-linear equation required 4260 time steps with an average $\Delta t = 2.3\text{e}{-7}$. The non-linear wave equation was solved in 1.26s with 531K allocations of total size 1.56 GiB. In the figures below the results are shown together

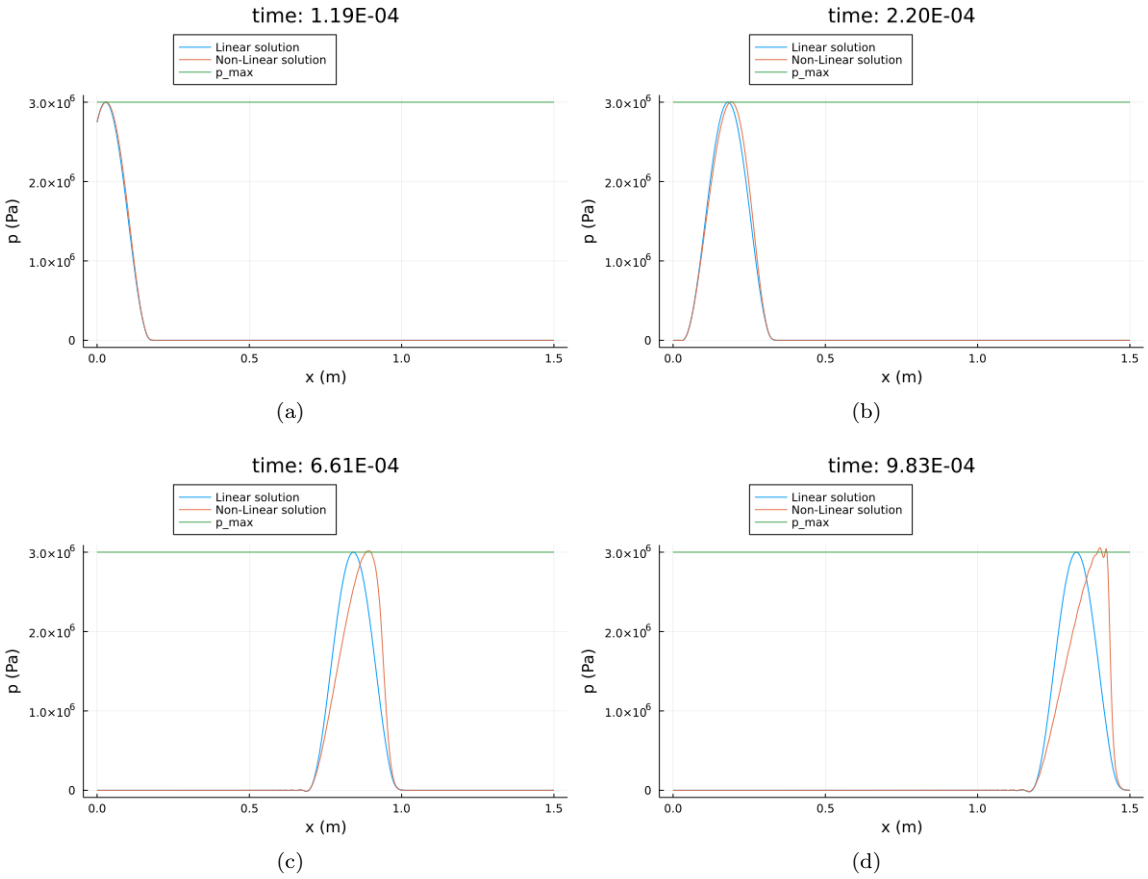


Figure 12: Plot of two simulations one with non-linearity and one without, shown at four different times.

In figures12(a-d) the effect of the non-linearity can clearly be seen. Additionally it can be seen that both waves do not decrease in power as they move along the x-axis. The top of the wave has a higher pressure and thus a higher wave speed compared to the bottom of the wave. This results in the top of the wave overtaking the first parts of the wave forming a shock wave. In figure a the wave is still starting to form, in figures b and c the shock front continues to grow and becomes steeper. In figure d the shock wave is almost vertical, also irregularities can be seen at the top of the wave. One of the assumptions made by FDM is that the function is smooth enough. This is perfectly the case for the linear wave but for the non-linear wave the shock front will ultimately form an almost discontinuity which causes the FDM assumption to no longer be valid, and thus start causing errors. For a model with implemented attenuation this would be less of a concern since the attenuation will dampen the shock front.

This inability to handle shock fronts is a major downside of our model and/or implementation. When specifying a model we always have to take care not to let the shock front become too steep, since this results in the FDM failing to return a solve. This failure of our model will be more deeply discussed in chapter 7.

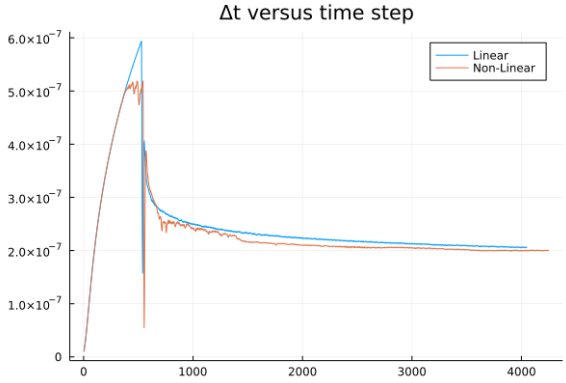


Figure 13: Plot showing the size of each time step for both simulations. For readability the data was made smoother by use of a moving average of 10 data points.

In the above figure the size of time steps are shown. A greater size of time step means the solver is able to take bigger steps and thus solve the differential equation faster. It can be clearly seen that the solver is able to take bigger steps for the linear wave equation compared to the non-linear equation. Additionally the solver is in both cases able to take larger steps at the beginning of the simulation.

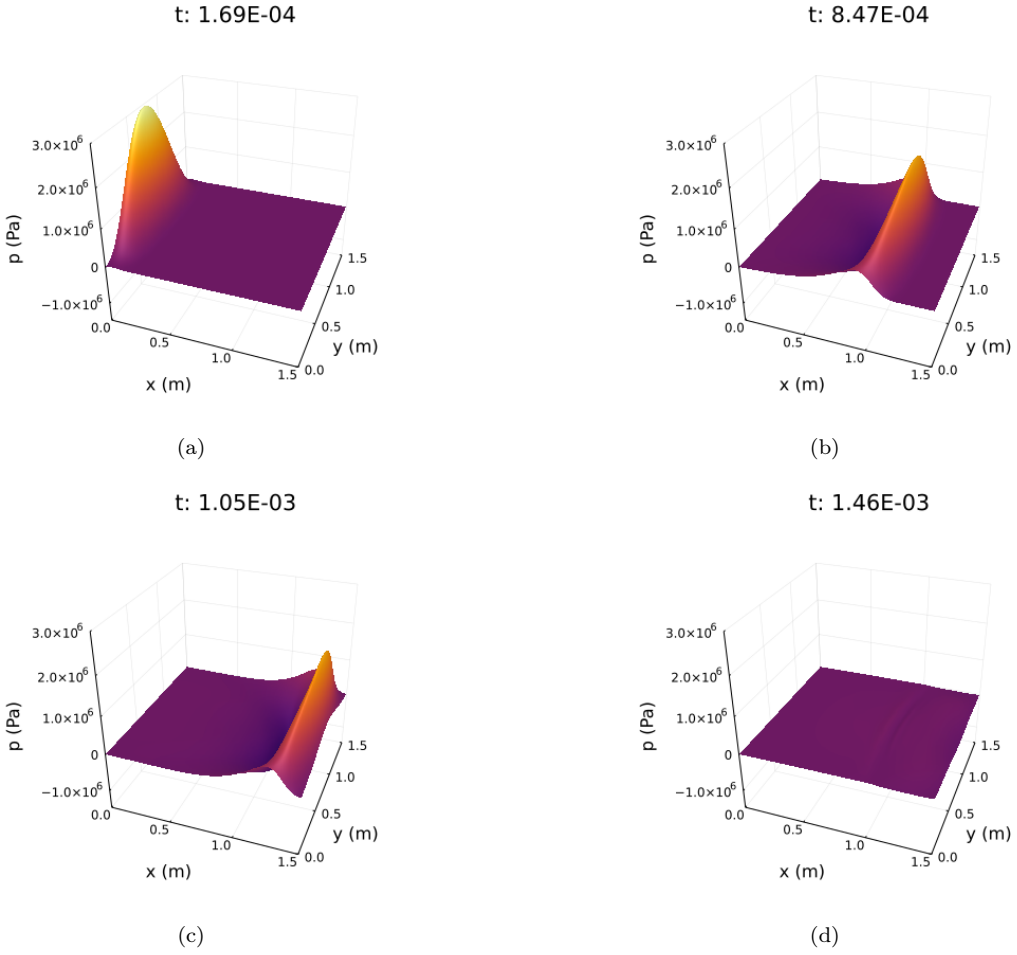
6.3 Driven Non-Linear Wave 2D

We will now show the results of our full 2D model as described in chapter 3. This model includes a driving side and 3 radiating sides, with non-linearity. We again used the constants as in table 1, with T = 1ms, N = 150 giving a grid of $150 \cdot 150 = 22500$ total points. The driving function was defined as

$$f(t,y) = sin(\pi\alpha \cdot t)^2 sin(\frac{\pi}{L}y)^2 \qquad \alpha = 10^4, t < 1/\alpha$$

$$f(t,y) = 0 \qquad t > 1/\alpha$$

With the choice of squared sines stemming from the fact that transition to squared sin from a zero function is continuous in first and second derivative. As a solver we manually chose the non-stiff solver Tsit5, this choice will be expanded on in chapter 7. Which solved the stated problem in ≈ 19 seconds with 1.03M allocations of total size 3.458 GiB. This large number of allocations caused approximately 69% of the total computation time to be spend on garbage collection. The solver required 714 time steps with an average $\Delta t = 2.8\text{e}{-6}$.The results can be seen in the following figures



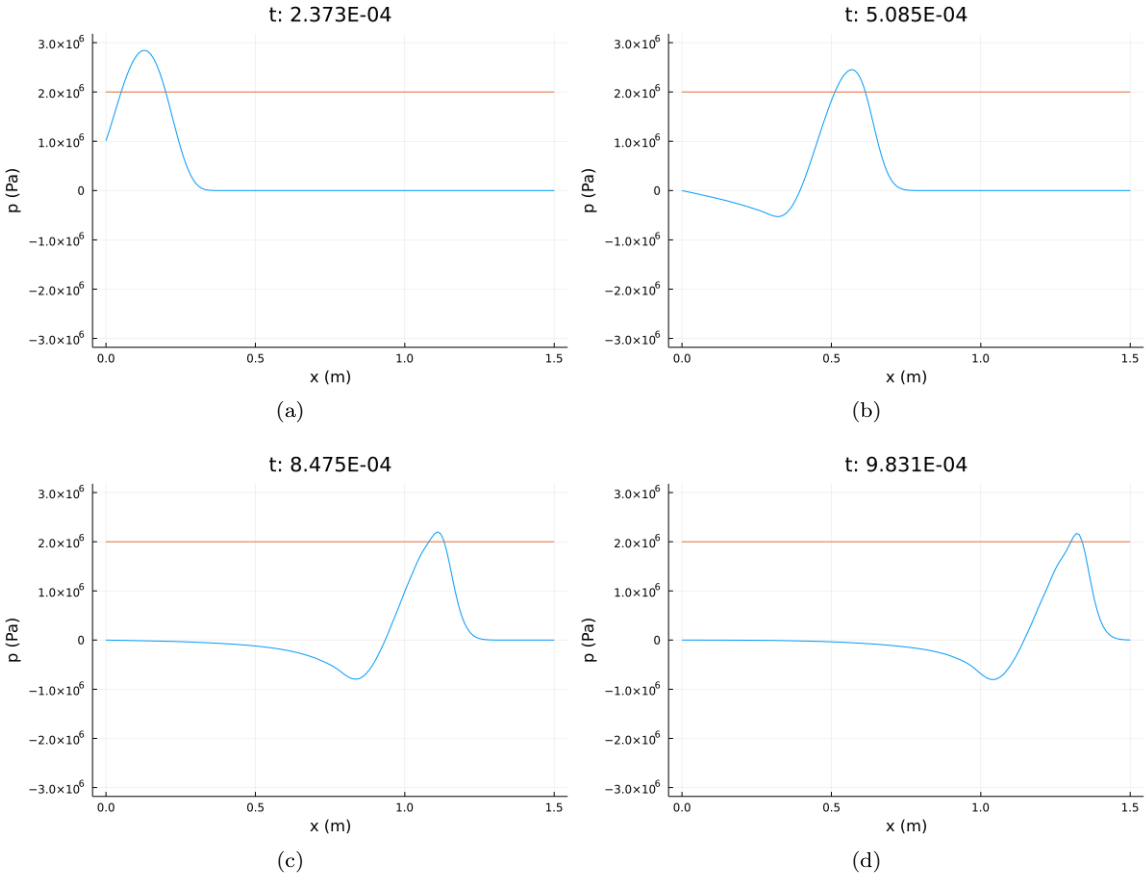
(a) (b)

(c) (d)

Figure 14: Plot of the 2D non-linear wave solution, shown at four different times.

In the above figures are shown the solution to the 2D non-linear wave equation for different times. In 2D it is harder to visualize what is happening in the solution so i advice to look at the animations on the GitHub page. Nonetheless it can be seen in the figures the wave is traveling forward and radiating at all the boundaries. Furthermore the reflection at the boundaries is not non-existent but it is small, fully non reflecting boundaries will be improbable with the use of FDM due to approximation errors. The non-linearity is less visible here in this model but it does effect the wave equation.

To get better understanding of how the wave deforms as it travels we will now look at a slice in the y direction at $y = 0.5,\ L = 0.74m$. This slice is shown in the below figures for different values of t, a guide line is added at amplitude $\frac{2}{3}P_0$ to aid in understanding how the top of the wave behaves.



(a) (b) (c) (d)

Figure 15: Plot of a slice of the 2D non-linear wave solution at $x = 0.5L$, shown at four different times.

In the figures above the movement of the non-linear wave along a slice in the x direction can be seen. The effect of non-linearity can be seen with the formation of a shock front, along with the tail end of the wave slowing behind. Additionally a dip can be seen which forms behind the wave, it is unclear if this is caused by the non-linearity by numerical errors or by an implementation error. Lastly the top of the wave shrinks as it moves further along the y axis, while it would be expected for top to grow larger caused by the non-linearity. The reason it does drop as it moves is because the wave also moves in the x direction. This causes it to dissipate energy in directions other than the x-axis, these dissipating waves are then absorbed by the radiating boundary conditions as if they would move towards infinity.

6.4 Initial Conditions Non-Linear wave 1D

The final model we will show is that of the 3D non-linear wave equation as stated in chapter 3, with nonzero initial conditions along with homogeneous boundary conditions. For this the following initial conditions are used:

$$f(x,y,z) = sin(\frac{3\pi x}{L})^4 sin(\frac{3\pi y}{L})^4 sin(\frac{3\pi z}{L})^4 \qquad x,y,z \in [\frac{L}{3}, \frac{2L}{3}]$$
$$f(x,y,z) = 0 \qquad otherwise$$

This initial function was chosen to achieve a sharp peak around the center. The initial conditions is shown in the following figure

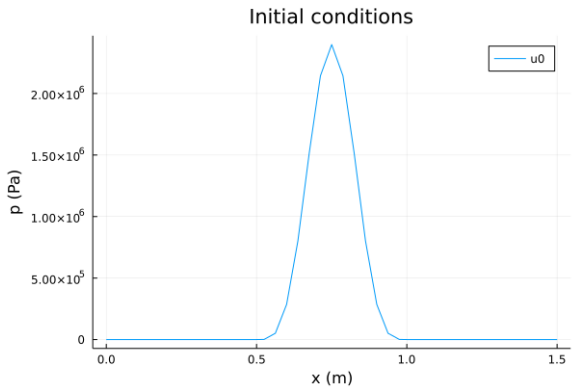


Figure 16: Plot showing the initial conditions used for our model of the 3D non-linear wave equation.

For this model we again used the constants as described in table 1, with T = 2ms and N = 40. Resulting in a grid of $40^3 = 64000$ points. The solver was left unspecified, the algorithm then chose the non-stiff Tsit5 solver. The acceptable absolute and relative error was again set at $\epsilon = \text{e}{-6}$. The solve was successful in ≈ 11 seconds, with 5.54k allocations of total size 2.684 GiB. Around 22% of total computation time was spent on garbage collection. The solver required 236 time steps, resulting in an average $\Delta t = 8.5\text{e}{-6}$. Due to the difficulty of plotting the results in 3D we will show the results in three sets of figures. First we will show a 3D plot of the solution shown at 4 different times

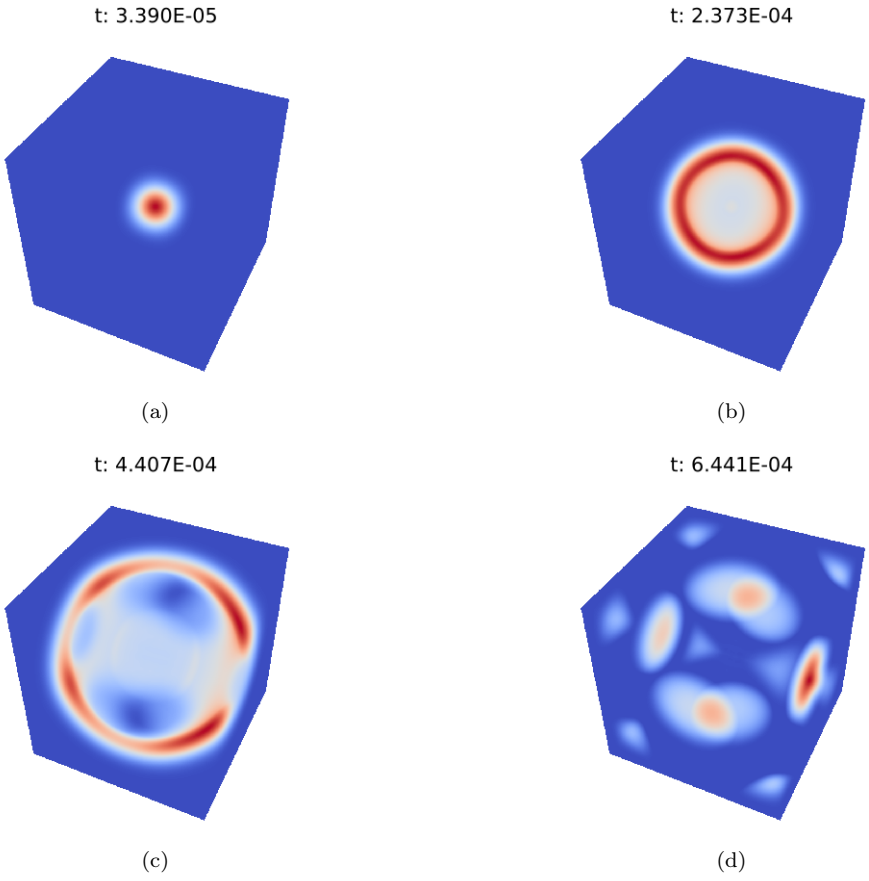


Figure 17: Plot of the 3D non-linear wave solution, shown at four different times. Using a Maximum Intensity Projection technique.

In the figures above the wave can be seen expanding from the center. With the wave bouncing of the edges as can be seen in figure 17(d). The plotting technique used does not allow for the imaging of negative values, thus for later times this technique is not useful. Instead we will now look at a slice of the solution. Specifically we will look at the slice $(x, y, L/2)$, the results are shown in the figures below.

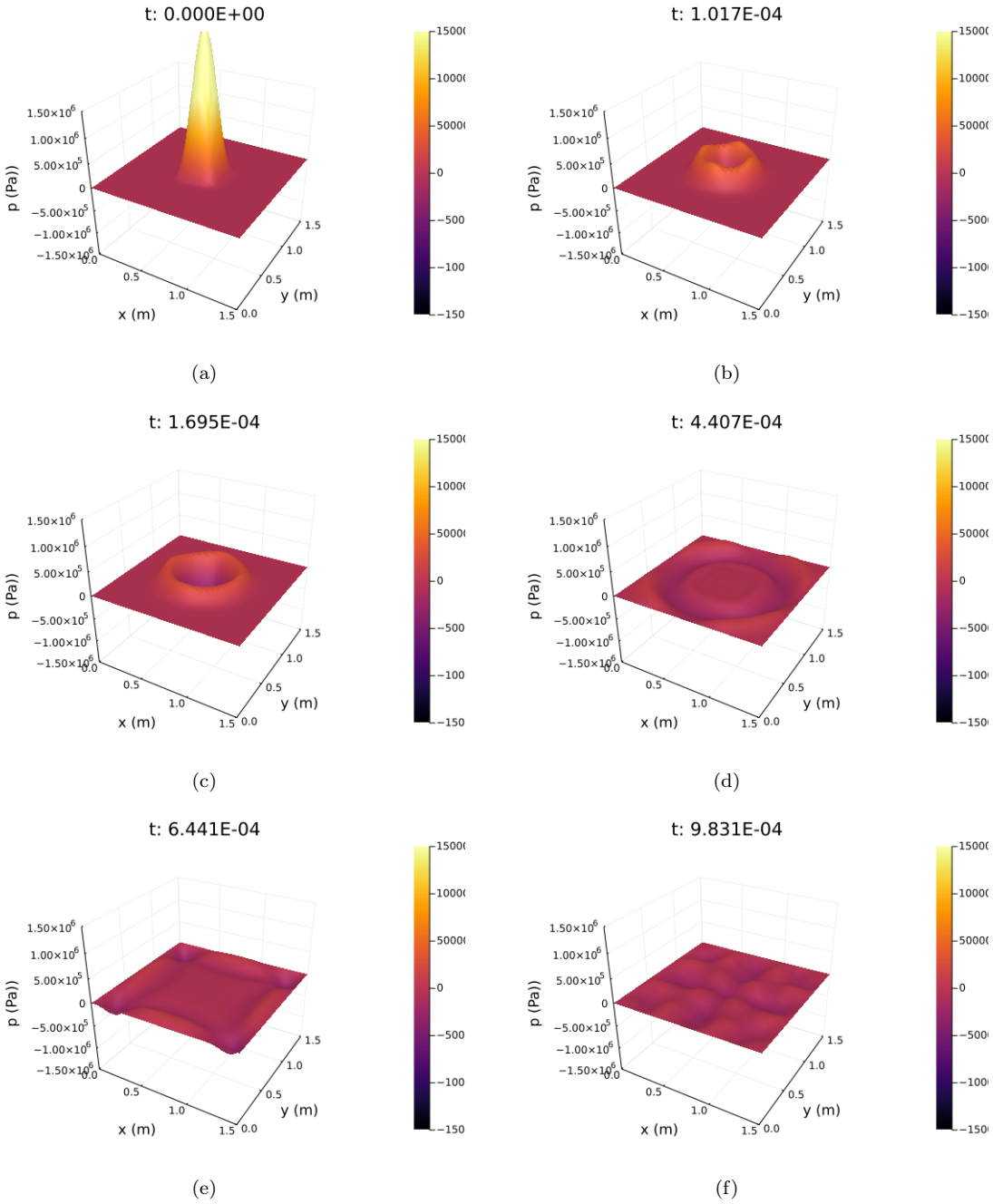


Figure 18: Plot of a slice of the 3D non-linear wave solution, shown at 6 different times.

In the above figures a slice around the middle of the solution is shown. In the first figure the initial conditions used can be seen. Thereafter the wave can be seen to start moving away from the center in all directions, making a sort of bowl form with a dip in the middle. In figure (c) the wave is expanding further outward, by now the wave has decreased in intensity significantly. In figure (d) the wave has partially reached the boundary, and started reflecting. Then in figure (e) the wave has reflected from the boundary, now returning to the center with opposite sign. In figure (f) the reflected wave again reaches the center. At some parts the wave function is again positive. This is because the wave reflected twice, ones in the x direction and once in the y direction. This causes the wave to invert twice, thus returning to its original sign. Now a great deal of interference can be seen with a complex pattern being formed. At times greater than the ones shown it is hard to discern what is occurring in the plot. This is because of all the reflections including those in the z direction interfering. This makes for a chaotic looking plot for which it is difficult to make observations. The non-linearity is not clearly visible here. A full animation can be found on my GitHub.

Finally we will look at a single line of our domain. Namely the line (x,L/2,L/2). This will allow us to get a look at the effect of non-linearity on the model. The results are shown in the figures below

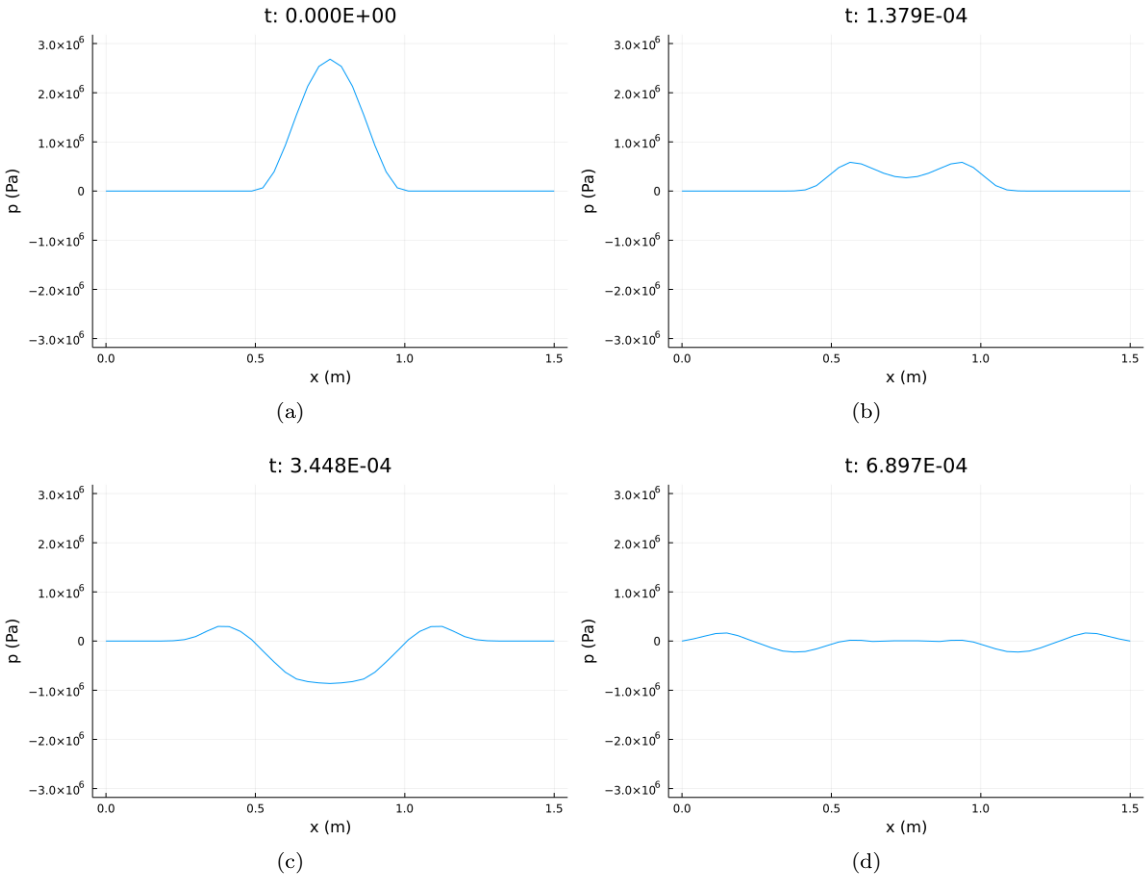


Figure 19: Plot of a line withing the domain of the 3D non-linear wave solution, shown at 4 different times.

Here in the first plot again the initial condition can be seen. In the second plot the wave can be seen to separate in a left moving and right moving part as expected. In figure (c) the wave also becomes negative in between the positive right and left moving parts, this is not expected behaviour for the linear wave equation. Thus we expect this is because of the non-linearity. Additionally this behaviour was not seen in our simulations of the linear wave equation with similar initial conditions, leading us to not think it is a result of our implementation. In figure (d) the wave has traveled even further in both directions and has decreased in intensity significantly. Because of the decrease in intensity an effect of non-linearity is not clearly visible here. The decrease of intensity can be falsely attributed to attenuation, but remember that we have not implemented attenuation in our model. The decrease in intensity actually comes from the dissipation of the wave in other dimensions. In a 1D model the wave can only move in two directions so we expect the right and left moving parts to be approximately half of the original intensity. In 3D however the wave can move in 3 dimensions and in even more directions, leading the wave to decrease in intensity rapidly. Because the wave is quickly reducing in intensity the effects of the non-linearity do not create a shock front.

7 Conclusion and Discussion

In this thesis we have successfully implemented a FDM solution to the multi-dimensional Westervelt equation without attenuation. We have produced multiple models showcasing the Westervelt equation for different conditions, including a three dimensional non-linear model. As well as a two dimensional model which is driving on one side as is the case in a real ultrasound transducer. Additionally we have verified the accuracy of our implementation in a one dimensional domain, by comparing our solution of the linear wave equation to the known analytical solution. Lastly we have also made a comparison between our solutions for the one dimensional non-linear wave equation with the one dimensional linear wave equation.

We were unsuccessful in implementing attenuation into our models. Since the attenuation term is a 3rd order derivative in time, we have to define a system of 3 coupled differential equations. When we tried implementing attenuation into our models, we were hampered by instability. Even for the simple one dimensional non-zero initial condition model, it was difficult to achieve a stable solution. None of the non-stiff solvers provided with DiffEq.jl were able to solve this system, and only some stiff solvers were able to solve the system. However these stiff solvers also failed when slightly increasing the non-linearity coefficient. The stiff solvers where only able to solve the system when we used a unrealistically low non-linearity coefficient. This is in stark contrast to the models without attenuation, for these models virtually any solver provided by DiffEq.jl was able to solve the system. Since the attenuation was causing severe instability in our implementation even in a simple model, we decided to not pursue adding attenuation in our higher dimensional models and not include it in our results.

In the one dimensional models our implementation applied to the linear wave equation closely matched the known analytical solution, although there was a small discretization error which could be a result of our method used or of an error in our implementation. When implementing the non-linear wave equation to analyze the behaviour of non-linearity our results matched that of previous students. Additionally in this model the downside of not including attenuation was clearly visible, because the wave front became to steep which led to numerical errors arising at the top of the wave. These errors would accumulate until the whole system became unstable. If attenuation was included it would dampen the wave front before becoming to steep. This would allow us to worry less about the system becoming unstable as a result of a shock front forming.

In our main two dimensional model we successfully implemented a model closely resembling that of a medical transducer. With one driven and multiple radiating sides. To further improve this model we would advice analyzing a model in which there is a variance in material somewhere in the domain caused by an object being studied. This would allow study of the reflections associated with a change in medium. And possibly an attempt at reconstructing the object from the reflected waves. For this its possible FEM would be advantageous as it could more easily increase the mesh density around a change in medium.

In our three dimensional model we successfully implemented the non-linear wave equation with homogeneous boundary conditions and non-zero initial conditions. We were able to solve the 3D model in just 11 seconds thanks in part to the improvement of computation performance by Julia. We were able to study the effects of non-linearity in three dimensions. We noted a smaller effect of the non-linearity compared to our one dimensional mode. This is in part because the wave diffuses to quickly in three dimensions. With the wave expanding in all directions the intensity of the wave greatly decreases as the wave travels. This effect of diffusing in all directions was greater than the effects of non-linearity.

By using Julia and the DiffEq.jl package we were able to quickly implement various FDM methods and thus choose a appropriate solver for our different models. We had studied various methods meant to improve implicit solver performance, such as providing a analytical expression for the jacobian. But in the end all our models used a non-stiff solver. This is because the non-stiff solvers are generally much faster than their stiff counterparts. Especially in 3D where we had a grid of 64000 points for which we have to store the amplitude as well as the velocity, leading to 120.000 total values store. This massive grid was too computationally expensive for implicit solvers to be able to solve. And the non-stiff solvers did not suffer too greatly from instability caused by stiffness. We do not have a direct comparison between Julia and MATLAB or python which previous students used. But it is our believe that our implementation would have been more difficult in python/MATLAB, in our 2D model for example we devectorized large parts of the function. In other high-level languages this would lead to a massive slowdown, but in Julia this was as fast as a vectorized implementation.

Further Research

For further research we recommend to study the use of FDM for solving the Westervelt equation in contrast to FEM methods. First of all we can easily specify a regular domain for which the FDM has a easily implemented discretization. If the domain were to be irregular this would give reason to use FEM. With implementing FEM we think the previous students achieved a possibly better spatial discretization, but they then gave in by using a basic time solver which accrued a larger error attributable to the time integration. In this consideration by focusing more on lowering the error attributable to the spatial discretization or time integration, we think it is better to focus on lowering the error of the time integration. To make this statement more robust an in

depth analysis is needed to compare the errors attributable to the discretization or time integration. Finally the spatial discretization as used in our implementation can be made more advanced by using higher order finite difference approximations to the spatial derivatives, this does include sampling the function at more locations. These however do come at the cost of making the time integration more expensive. All together we think we have provided a meaningful argument for why FDM should be considered when solving the Westervelt equation.

References

[1] K.E. Atkinson. *An Introduction to Numerical Analysis*. Wiley, 1978.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[3] J. G. Charney, R. FjÖrtoft, and J. Von Neumann. Numerical integration of the barotropic vorticity equation. *Tellus*, 2(4):237–254, 1950.

[4] Deutsche Akademie der Wissenschaften zu Berlin. *Histoire de l'Académie royale des sciences et des belles lettres de Berlin*. A. Haude, 1749.

[5] Jonnie Diegelman. Modeling spacecraft seperation dynamics in julia, 2021.

[6] G. Wanner E. Hairer. Solving ordinary differential equations ii, stiff and differential-algebraic problems. *Computational mathematics (2nd revised ed.)*, 1996.

[7] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Math. Comp.*, 51:699–706, 1988.

[8] Mark F. Hamilton and D. T. Blackstock. *Nonlinear acoustics*. Academic Press, 1998.

[9] Nick Heath, TechRepublic, Andy Wolber, Lance Whitney, Cedric Pernet, Moira Alexander, and Veronica Combs. Why is programming language julia growing so fast and where is it going next?, Sep 2018.

[10] E.F.G. van Daalen J. Broeze. Radiation boundary conditions for the two-dimensional wave equation from a variational principle. *Mathematics of computation*, 58, 1992.

[11] W. Kutta. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeit. Math. Phys.*, 46:435–53, 1901.

[12] T. Lauterborn W., Kurz and I. Akhatov. *Nonlinear Acoustics in Fluids in Handbook of Acoustics*. Springer, 2007.

[13] Christopher Rackauckas. Ode solver multi-language wrapper package work-precision benchmarks (matlab, scipy, julia, desolve (r)), 2022.

[14] Christopher Rackauckas and Qing Nie. Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia, 2017.

[15] Christopher Rackauckas and Qing Nie. Confederated modular differential equation apis for accelerated algorithm development and benchmarking. *Advances in Engineering Software*, 132:1–6, 2019.

[16] C. Runge. Ueber die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46:167–178, 1895.

[17] Ch Tsitouras. Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, 62(2):770–775, 2011.

[18] James H Verner. Numerically optimal runge–kutta pairs with interpolants. *Numerical Algorithms*, 53(2-3):383–396, 2010.

[19] Marcella Zijta. Solving the westervelt equation with losses using first and second order finite element method, 2017.

Appendices

A 2D Discretization derivation

In 2D the space dimensions will be discretized with as a NxN grid of nodes Since the domain is LxL, this gives $\Delta x = \Delta y = \frac{L}{N}$ We will now define $u[n_x, n_y](t) = u(n_x \cdot \Delta x, n_y \cdot \Delta y, t)$

$$\nabla^2 u(x,y,t) = \frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial x^2}$$

We use the same 2nd order central difference for the second derivative, which gives

$$\nabla^2 u[n_x, n_y](t) = \frac{u[n_x - 1, n_y](t) - 2u[n_x, n_y](t) + u[n_x + 1, n_y](t)}{\Delta x^2} + \frac{u[n_x, n_y - 1](t) - 2u[n_x, n_y](t) + u[n_x, n_y + 1](t)}{\Delta y^2}$$

Since the full 2D laplacian as stated above is cumbersome to write i will leave the laplacian as ∇^2. Substituting the 2D discretization in the Westervelt equation gives us the following

$$\frac{\partial^2 u[n_x, n_y](t)}{\partial t^2} = \nabla^2 u[n_x, n_y](t) - \frac{2\beta}{\rho_0 c^2}\left(u[n_x, n_y](t)\frac{\partial^2 u[n_x, n_y](t)}{\partial t^2} + \left(\frac{\partial u[n_x, n_y](t)}{\partial t}\right)^2\right)$$

B 3D Discretization derivation

In 3D the derivation is similar to the 1D and 2D derivations so i will not expand too much on the derivations. We use a grid of NxNxN nodes.
The domain is LxLxL which gives $\Delta x = \Delta y = \Delta z = \frac{L}{N}$
We define $u[n_x, n_y, n_z](t) = u(n_x \cdot \Delta x, n_y \cdot \Delta y, n_z \cdot \Delta z, t)$ Again using the 2nd order central difference for the second derivatives,which gives

$$\begin{aligned}\nabla^2 u[n_x, n_y, n_z](t) = &\frac{u[n_x - 1, n_y, n_z](t) - 2u[n_x, n_y, n_z](t) + u[n_x + 1, n_y, n_z](t)}{\Delta x^2} \\ &+ \frac{u[n_x, n_y - 1, n_z](t) - 2u[n_x, n_y, n_z](t) + u[n_x, n_y + 1, n_z](t)}{\Delta y^2} \\ &+ \frac{u[n_x, n_y, n_z - 1](t) - 2u[n_x, n_y, n_z](t) + u[n_x, n_y+, n_z + 1](t)}{\Delta z^2}\end{aligned}$$

Which finally gives the following discretized Westervelt equation

$$\frac{\partial^2 u[n_x, n_y, n_z](t)}{\partial t^2} = \nabla^2 u[n_x, n_y, n_z](t) - \frac{2\beta}{\rho_0 c^2}\left(u[n_x, n_y, n_z](t)\frac{\partial^2 u[n_x, n_y, n_z](t)}{\partial t^2} + \left(\frac{\partial u[n_x, n_y, n_z](t)}{\partial t}\right)^2\right)$$