Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Comparison of Implementation Strategies for PDE Adjoint Problems

Rebecca Elisabeth Brydon

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Comparison of Implementation Strategies for PDE Adjoint Problems

Author:	Rebecca Elisabeth Brydon
1st examiner:	Univ.-Prof. Dr. Michael Bader (TUM)
Assistant advisor:	Lukas Krenz (TUM)
Assistant advisor:	Dr. Anne Reinarz (Durham University)
Submission Date:	June 15th, 2021

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

June 15th, 2021 Rebecca Elisabeth Brydon

Acknowledgments

I would like to thank my supervisors Dr. Anne Reinarz (Durham University) and Lukas Krenz (Technical University Munich) for their support and supervision! I would also like to thanks Prof. Micheal Bader of the to the Scientific Computing department for giving me the opportunity to write my master thesis at the research chair.
A second big thank you to Anne Reinarz, for helping me apply as a visiting student at Durham University. Many thanks to the Computer Science Department at the University of Durham for giving me the opportunity to visit Durham University.

Irrtümer haben ihren Wert; jedoch nur hier und da. Nicht jeder, der nach Indien fährt, entdeckt Amerika.

-Erich Kstner

Abstract

Common problems in seismology are: detecting the depth of soil layers, approximating the material parameters or determining the origin of an earthquake. These parameters can be determined by solving an *Optimal Control Problem*. In this thesis, we consider Adjoint Problems and how they can be used for parameter optimizations. As a gradient based optimization routine would require the first derivative of the cost function with respect to the optimization parameter [1]. A cost function J computes the cost as the norm between the measurements and a simulated solution which is dependent on the model parameters. As the cost function is a composite function and depends on the PDE parameters α and also on the PDE solution, the gradient $DJ/D\alpha$ can not be computed so easily [2]. It is the second dependency makes things more complex, as in most cases there is no functional representation of the PDE solution which would be directly differentiated. A solution is to compute the derivative via the adjoint equation of the PDE [1]. In this thesis two approaches for computing the total derivative are compared. The first approach *Differentiate-then-discretize* derives the adjoint equation analytically and then is discretized using the DG method [1]. The second ansatz, *Discretize-then-Differentiate*, uses Automatic Differentiation (AD) to differentiate an already discrete forward solver [1]. We create two prototype implementations of both adjoint solver approaches in the Discontinous Galerkin solver TerraDG [3]. For comparison, we also implement the hyperbolic model equations in FEniCS, an open-source finite element solver which provides a python interface [4], [5]. The dolfin-adjoint package [6] can then be used to automatically compute the gradient via the adjoint.

Contents

Part I.

Introduction

1. Introduction

1.1. Inverse Problems in Wave propagation

In science we often make some observations of a physical phenomena and record these as measurement data. An Inverse Problem (IP) arises, if we want to determine the parameters which caused the phenomena but can not be measured. In this thesis we consider Adjoint Problems and how they can be used for parameter optimizations.

Common problems in seismology are: detecting the depth of soil layers, approximating the material parameters or determining the origin of an earthquake. These parameters can be determined by solving an *Optimal Control Problem*. A very good introduction to Optimal Control Problems in Flow Optimization is given in [1]. For example, assume the origin (s_x, s_y) of an earthquake is to be determined. We have recorded some measurements of the displacement or the stress. A gradient based optimization routine would require the first derivative of the cost function with respect to the optimization parameter [1]. A cost function J computes the cost as the norm between the measurements and a simulated solution which is dependent on the model parameters. As the cost function is a composite function and depends on the PDE parameters α and also on the PDE solution, the gradient $DJ/D\alpha$ can not be computed so easily [2]. It is the second dependency makes things more complex, as in most cases there is no functional representation of the PDE solution which would be directly differentiated. A solution is to compute the derivative via the adjoint equation of the PDE [1].

In this thesis two approaches for computing the total derivative are compared. The first approach *Differentiate-then-discretize* derives the adjoint equation analytically and then is discretized using the DG method [1]. However, the derivation of the adjoint equation can quickly become quite complex and has to be derivation specifically for the considered PDE. This is why it is interesting to look at a second ansatz: *Discretize-then-Differentiate*. In this case we use Automatic Differentiation to differentiate an already discrete forward solver [1]. The advantage is that the AD computation can be implemented generically for all PDE equation of a certain form. On the other hand the computational cost for computing the gradient with AD are however quite high. We create two prototype implementations of both adjoint solver approaches in the Discontinous Galerkin solver TerraDG [3]. For comparison, we also implement the hyperbolic model equations in FEniCS, an open-source finite element solver which provides a python interface [4], [5]. The dolfin-adjoint package [6] can then be used to automatically compute the gradient via the adjoint. For testing, a parameter optimization example is setup following the general structure of the dolfin-adjoint tutorials [7].

Part II.

Theory: Adjoint equations in Optimal Control Problems

2. Hyperbolic wave problems

A general formulation of a hyperbolic partial differential equation (PDE) is given in [8] as

$$\frac{\partial \phi}{\partial t} + \nabla \cdot \mathcal{F}^c(\phi) = f, \tag{2.1}$$

where ϕ is the state vector and $\mathcal{F}^c(\phi)$ represents the physical flux. In the following we use the general Discontinuous Galerkin (DG) formulation $\mathcal{N}(u,v)$ given [9] for a convective-diffusive system. As the above PDE system is only convective, we adapt the general formulation by setting the viscous terms in $\mathcal{N}(u,v)$ to be zero. We also disregard the interior penalty term for simplicity. The reduced DG-formulation then is

$$\begin{aligned}
\mathcal{N}(u,v) = &- \int_\Omega \mathcal{F}(\phi)\nabla v dx - \int_\Omega f \cdot v \, d\Omega && (2.2)\\
&+ \sum_{\kappa \in \mathcal{T}} \int_{\Gamma \backslash \partial\Omega} \mathcal{H}(\phi, n^+) \cdot v^+ \, ds && (2.3)\\
&+ \int_\Gamma \mathcal{H}(\phi, gD, n^+) \cdot v \, ds = 0.
\end{aligned}$$

As defined in [9], the domain Ω with the boundary $\Gamma = \partial\Omega$ is discretized on a mesh $\mathcal{T}$ with the elements κ where $\partial\kappa$ is the boundary of an element and n_k is the outward pointing normal on a face. The numerical flux $\mathcal{H}$ is approximated using the local Lax-Friedrichs flux given in [9] defined as

$$\mathcal{H}(\phi, n^+) = \{\{\phi\}\} \cdot n^+ + \frac{1}{2}\lambda_{max} \cdot [[\phi]], \tag{2.4}$$

where $\{\{\phi\}\} = \frac{1}{2}(\phi^+ + \phi^-)$ is the average and $[[\phi]] = \phi^+ - \phi^-$ is the jump [9].

2.1. Test equations

2.1.1. Advection equation

The simplest hyperbolic wave equation is the linear advection equation. We use the definition for a constant flow given in [10]

$$\begin{aligned}
\frac{\partial \phi}{\partial t} + \beta \cdot \nabla \phi &= 0 && \text{in } \Omega \times (0,T) && (2.5)\\
\phi(t, \mathbf{x}) &= \phi_0(\mathbf{x} - \beta t) && \text{in } \partial\Omega \times (0,T)\\
\phi(0, \mathbf{x}) &= \phi_0(\mathbf{x}) && \text{in } \Omega \text{ at } t = 0
\end{aligned}$$

where the flow velocity in two dimensions is assumed to be the vector $\beta = [\beta_x \quad \beta_y]^T$. In one dimension the flow velocity becomes a scalar [8]. The initial condition at $t = 0$ over the domain Ω is given by a function ϕ_0, depending only on $\mathbf{x}$. The boundary condition are defined using the exact solution, which can be computed according to [8], as the initial solution shifted by the flow velocity depending on the current time t. An example initial condition in two dimensions is the mock one-dimensional setting $\phi_0(\mathbf{x}) = sin(2\pi x)$.

2.1.2. Linear elastic wave equations

As a more advanced example equation, we consider the linear elastic wave equation, as it is a common base model used in seismology [8]. To better understand the physical forces considered by the model, we look at the simple example given in [8] of a two dimensional beam. If the beam is compressed at $[x = 0, y]$, the force will cause a "pressure" wave, often shorted to P-wave, to propagate along the x-axis with the wave speed c_p. As the equation system couples the stress ϕ and the motion $\dot{u}$, the P-wave will accelerate the wave motion. The model however only holds if the deformations inside a solid are small, as in this case stress and strain can be linearly related by Hook's law [8]. For compression forces, the behaviour of the elastic wave in solids is analogous for acoustic waves in gas or liquids. However, on solids there is a second type of force: the shear force. If the solid is displaced in y-direction, the material will show small elastic deformations. As long as the deformations are small enough, the bonds between the molecule of the solid will try to restore the original state. The resulting waves are called S-waves and move orthogonality to the direction of wave motion with wave speed s_s. The equation system is only linear, if the deformations are linearly proportionally to the restoring forces. If the material is over stretched to breaking point, the resulting behaviour would clearly no longer be linear.

The elasticity equation system is given in [11] in second order form as

$$\begin{aligned} \rho \frac{\partial^2 u}{\partial t^2} - \nabla \sigma = 0 \quad & \text{in } \Omega \times (0, T) \\ \sigma \cdot n = T \quad & \text{in } \Gamma \times (0, T) \\ u = u_0(\mathbf{x}) \quad & \text{in } \Omega\text{, t=0} \end{aligned} \tag{2.6}$$

which relates the displacement u, the stress σ and the acceleration $\frac{\partial^2 u}{\partial t^2}$. The material density is given by ρ. As DG formulations are generally derived for first order PDE system, we consider the first order derivations and equation definitions given in [8]. The three dimensional elastic wave equation can be reduced to a two dimensional plane if the variations in the third direction is zero. The five equations (2.7), defined in [8], can be used to model the motion of S-waves and P-waves in a two dimensional plane.

$$\frac{\partial}{\partial t}\begin{bmatrix}\sigma^{11}\\ \sigma^{22}\\ \sigma^{12}\\ u\\ v\end{bmatrix} - \frac{\partial}{\partial x}\begin{bmatrix}(\lambda+2\mu)u\\ \lambda u\\ \mu v\\ \frac{1}{\rho}\sigma^{11}\\ \frac{1}{\rho}\sigma^{12}\end{bmatrix} - \frac{\partial}{\partial y}\begin{bmatrix}\lambda v\\ (\lambda+2\mu)v\\ \mu u\\ \frac{1}{\rho}\sigma^{11}\\ \frac{1}{\rho}\sigma^{22}\end{bmatrix} = 0. \tag{2.7}$$

For some simpler setups, e.g. the planar wave problem given in [12], it is still possible to find an analytic solution which can be applied at the boundary. When computing a seismic simulation, for example the simple LOH1 earthquake scenario, a physically motivated boundary condition must be applied [8]. We follow the boundary condition approach given in [8], where the exterior ghost cells out side the domain are set based on the interior cells in a physically meaning full way. This means for the ghost layer at the boundary we must ether specify a motion, i.e. set the velocities u, v or specify a traction T_{bnd} by setting the components of the stress in normal direction (outward pointing) according to

$$\begin{aligned} T &= \sigma \cdot \vec{n} \\ T_{out} &= 2 \cdot T_{bnd} - T_{in} \\ u_{out} &= u_{in} \\ v_{out} &= v_{in}. \end{aligned} \tag{2.8}$$

For the left and right boundary this means setting σ^{11} and σ^{12}, for the top/bottom boundary $T = [\sigma^{22} \quad \sigma^{12}]^T$. The stress perpendicular to the normal vector, σ^{11} for $n = (0,1)$ and σ^{22} for $n = (1,0)$, can not be controlled. If $T = 0$, then the boundary is called a *traction free boundary* and the velocities are simply copied from inside the domain [8].

Planar wave problem in one dimension

The two dimensional system can be reduced to a single dimension as shown in [8], by setting setting the displacements in y-direction to zero. The two decoupled one-dimensional systems, model the P-waves in x-direction and S-waves in y-direction separately [8]. As an test case, use the system derived for the displacements in y-direction in [8] as

$$\begin{aligned} \frac{\partial \sigma^{12}}{\partial t} - \mu \frac{\partial v}{\partial x} &= 0 \\ \frac{\partial v}{\partial t} - \frac{1}{\rho}\frac{\partial \sigma^{12}}{\partial x} &= 0, \end{aligned} \tag{2.9}$$

with the wave speed c_s and the analytical solution given in [13] as

$$\begin{aligned} v(x,t) &= \frac{1}{2}(sin(2\pi(x + c_s t)) + sin(2\pi(x - c_s t))) \\ u(x,t) &= \frac{1}{2}(sin(2\pi(x + c_s t)) + sin(2\pi(x - c_s t))). \end{aligned} \tag{2.10}$$

Planar wave problem in two dimension

We use the three dimensional planar wave problem given in [12] and reduce it to a two dimensional domain $\Omega = [0,1]^2$. The given analytical definition of the displacement in three dimension, is dependent on the three unit vectors p, d_p and d_p. By setting the wave propagation direction $\mathbf{p} = \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$ to move in x-direction instead of z, the components in the third dimension fall away. The P-Wave is set the same way, as it must run parallel with $\mathbf{d}_p = \mathbf{p} = \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$ [12]. The S-Wave propagates tangentially and therefore stays $\mathbf{d}_s = \begin{bmatrix}0 & 1 & 0\end{bmatrix}^T$. Inserting p,d_p and d_s into the analytic displacement definition, gives

$$\delta^1 = cos(k(x - c_p t)) \qquad \delta^2 = cos(k(x - c_s t)) \tag{2.11}$$

ans we can derive the analytical solutions for the velocities by differentiating δ w.r.t. to t giving

$$u = c_p k sin(k(x - c_p t)) \qquad v = c_s k sin(k(x - c_s t)). \tag{2.12}$$

For this test example, we use the analytical solutions to periodic boundary conditions. However, as we use the elastic wave equation in the stress-velocity formulation, we must also compute an analytic definition for the strain. For this we can use the relation given in [8]:

$$\epsilon = \frac{1}{2}[\nabla\vec{\delta} + (\nabla\vec{\delta})^T] = \begin{bmatrix} -k\sin(k(x - c_p t)) & -0.5k\sin(k(x - c_s t)) & 0 \\ -0.5k\sin(k(x - c_s t)) & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{2.13}$$

As the only non-zero strain components are ϵ^{11} and ϵ^{12}, the equation system [(22.11),[8]] can be simplified to compute the x and y stress components as

$$\begin{aligned} \sigma^{11} &= \epsilon^{11}(\lambda + 2\mu) \\ \sigma^{22} &= \epsilon^{11}\lambda \\ \sigma^{12} &= 2\epsilon^{12}\mu. \end{aligned} \tag{2.14}$$

2.1.3. Benchmark problem: Layer-over-halfspace 1 (LOH1)

A common model for simulating a basic earthquake scenario is the Layer-over-Halfspace 1 (LOH1) setup. The three dimensional setup given by [14] is used in the following but reduced to two dimensions along the y-z-plane. The domain dimensions are used for the FEniCS model as specified in the setup. For TerraDG the domain Ω is extended in z-direction to be come square, as non square domains are not jet supported. Figure **??** shows how the domain is divided into two sub-domains: a shallow *layer* on top of a large *half-space*. In the setup given in [14] an initial disruption is given by a point source which is located at twice the layer depth below the surface (top edge). An alternative way to

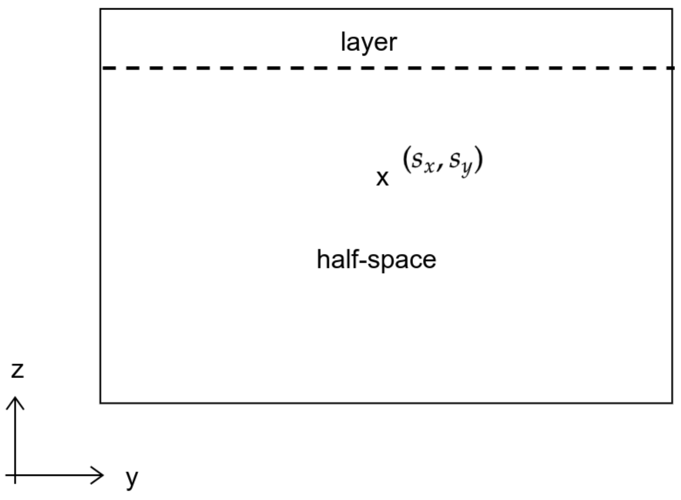


Figure 2.1.: The LOH1 setup following: The domain is subdivided into a layer region, with a 1km depth and a large half-space below [14]. The origin point of the source is located 2km below the top surface .

introduce an initial disruption without a point source is to set an initial stress in the two normal directions σ_{11} and σ_{11} in form of a tow dimensional Gaussian curve which has the center points (s_x, s_y). As the initial condition must satisfy the boundary condition given in (2.9), the Gaussian curve (2.15) is clamped to zero at the width L_x and height L_y of the domain.

$$\begin{aligned} \phi_0^{11} &= \exp^{(-0.01(x-s_x)-0.01(y-s_y)^2)}(L_x - x)x(L_y - y)y \\ \phi_0^{22} &= \exp^{(-0.01(x-s_x)-0.01(y-s_y)^2)}(L_x - x)x(L_y - y)y \end{aligned} \tag{2.15}$$

The remaining components ϕ^{12}, u and v are initialized with zero. As the domain dimensions and parameter settings are given in SI units, the problem length is scaled to $[km]$ with the scaling constant $SL = 10e3$. The flow speeds c_s and c_p are scaled analogously to $[km/s]$.

3. Adjoint equations in Optimal Control Problems

3.1. Optimal control problems

A common problem in many application fields of PDE models, is determining equation parameters based on a chosen cost function J, also called objective functional [1]. In the following section, we follow the general definitions given in [1] of an optimal control problem for a PDE F with state variables ϕ and the parameter α is given in [1] as

$$\begin{aligned} &\underset{\alpha}{\text{minimize}} && J(\alpha, \phi(\alpha), t) \\ &\text{subject to} && F(\phi(\mathbf{x}, \alpha), \alpha, t) \equiv 0, \; \mathbf{x} \in \Omega, \; t \in (0, T). \end{aligned}$$

The goal is to determine an optimal parameter value α_{opt} which minimize the cost function J, under the constraint that α_{opt} and corresponding solution vector ϕ_{opt} satisfy the PDE equation. For the optimization of the cost functional, a gradient based optimization routine can be used. However, these requires the gradient of the optimized function to determine the new search direction of the next step. A short summary of the symbols used in this chapter is given in Table (3.1).

Cost/objective function

A general formulation of a cost function which computes the cost over the complete time and space domain is given in [1] as

$$\begin{aligned} J_T(\phi, \alpha) = & \frac{\gamma_1}{2} \int_0^T \int_\Omega (\phi - \Phi)^2 d\Omega dt \\ & + \frac{\gamma_2}{2} \int_\Omega (\phi \mid_{t=T} - \Phi \mid_{t=T})^2 d\Omega + \gamma_s. \end{aligned} \tag{3.1}$$

We follow the explanation in [1], of the three terms of the cost functional to understand the motivation for choosing the given form. The first integral computes the difference between the solution ϕ and a measured solution Φ over the complete time and space domain. The second integral term is in principle already included in the first, as the time integral includes the final time T, it however improves the optimization results. The penalty term

$\gamma_s = \frac{\sigma}{2}\sum_{k=1}^{K}\alpha_k^2$ prevents the optimal control parameter value to become too large. An often used simplification of this cost function is obtained by setting $\gamma_1 = 0$, giving

$$J_C(\phi,\alpha) = \frac{\gamma_2}{2}\int_\Omega (\phi\mid_{t=T} - \Phi\mid_{t=T})^2 d\Omega + \gamma_s. \tag{3.2}$$

The time integral falls away and the cost is only computed using ϕ at the final time step. Equation (6.1) can also called a *controllability functional* [1]. In theory, one can compute the cost function at every node $x_{i,j}$ for $i,j = [1,\dots,N]$ of the discretized mesh. However, in practice it is simply infeasible to take so many measurements. We remind ourselves of the LOH1 setup, given in [14], where the displacement caused by an initial disruption are measured only at a few sensor points and not over the entire domain. We define the cost function then as

$$J_S(\phi,\alpha) = \frac{1}{2}\sum_{k=0}^{M}\sum_{i,j=0}^{S}(\phi(x_{i,j},t) - \Phi(x_i,t)^2 dt + \sum_{i,j=0}^{S}(\phi(x_{i,j},T) - \Phi(x_i,T)^2 dt, \tag{3.3}$$

where S is the number of sensor points and M the number of time steps.

3.2. Deriving the Adjoint equation

There are different approaches for deriving the first derivative i.e. the total derivative $\frac{DJ}{D\alpha}$ of the cost function J with respect to the equation parameters. The two main approaches can be divided into two categories, depending on the order in which the approach discretizes and differentiates the equations. Figure (3.1) visualizes the two different computation paths, which in the limit should converge to wards the same solution.

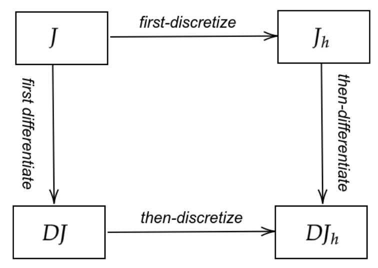


Figure 3.1.: Comparison visualization of the two approaches: first differentiate then discretize or vice versa.

Differentiate-first-then-discretize: The total derivative can then be computed by first setting up the Lagrangian for the optimal control problem and then differentiating with respect to

$F(\phi)$	PDE dependent on the state vector ϕ
Λ	Adjoint variable
J	cost / objection functional
α	general symbol for equation parameters
Ω	domain for $x, y \in [0,1]^2$
Γ	boundary of domain
t	time, $t_0 = 0$ start time and $T = 1$ end time

Table 3.1.: Symbols used for the definition of the optimal control problem and the derivation of the adjoint system.

the Lagrangian multipliers to obtain the adjoint equation. The cost function and the PDE with the corresponding adjoint equation is then be discretized.

Discretize-first-then-differentiate: The PDE equation system and the cost function are both discretized first and implemented in code. The differentiation to compute the adjoint equation is done using automatic differentiation (AD).

3.2.1. Deriving the adjoint using the Lagrangian function

In the following section, we derive the adjoint equation closely following the example derivation for the heat equation given in [1]. The Lagrangian function of an optimal control problem consists of four parts: the cost function, the PDE rearranged equal to zero, the boundary and initial conditions also rearranged to zero. As an initial example, the Lagrangian function with the multipliers Λ, τ and η is setup for the advection equation:

$$\frac{\partial \phi}{\partial t} + \beta \nabla \cdot \phi = 0 \quad \text{in } (0,T) \times \Omega, \tag{3.4}$$

$$\phi = g \text{ on } (0,T) \times \Gamma, \tag{3.5}$$

$$\phi = \phi_0 \text{ in at } t = 0. \tag{3.6}$$

The Lagrangian function for the PDE definition then is:

$$L(\phi, \alpha, \Lambda, \tau, \eta) = J(\phi, \alpha) \quad \underbrace{-\int_0^T \int_\Omega \Lambda \cdot \Big(\frac{\partial \phi}{\partial t} + \beta \nabla \phi\Big) d\Omega dt}_{\text{Integral over the PDE must be zero}} \tag{3.7}$$

$$\underbrace{-\int_0^T \int_\Gamma \tau \cdot \Big(\phi - g\Big) d\Gamma dt}_{\text{Integral over BC must be zero}} \tag{3.8}$$

$$\underbrace{-\int_\Omega \eta \cdot \Big(\phi \mid_{t=0} - \phi_0\Big) d\Omega}_{\text{Integral over IC must be zero}} \quad . \tag{3.9}$$

Setting the derivatives w.r.t to the Lagrangian multipliers Λ, τ and η equal to zero should recover the original PDE problem [1]. We can test this for the state equation by computing

$$\frac{dL}{d\Lambda} = \lim_{\epsilon \to 0} \frac{1}{\epsilon}\Big(L(\phi, \beta, \Lambda + \epsilon\tilde{\Lambda}, \tau, \eta) - L(\phi, \beta, \Lambda, \tau, \eta)\Big) \tag{3.10}$$

$$= \lim_{\epsilon \to 0} \frac{1}{\epsilon}\Big(- \int_0^T \int_\Omega \epsilon\hat{\Lambda} \cdot \Big(\frac{\partial \phi}{\partial t} + \beta\nabla\phi\Big) d\Omega dt\Big). \tag{3.11}$$

The variation ϵ cancels out and as we can choose $\hat{\Lambda}$ arbitrarily, the original PDE is recovered. To derive the adjoint equation we derivative of L w.r.t. the state variable and equal to zero:

$$\frac{\partial L}{\partial \phi} = \gamma_1 \int_0^T \int_\Omega (\phi - \Phi)\tilde{\phi} d\Omega dt + \gamma_2 \int_\Omega (\phi \mid_{t=T} - \Phi \mid_{t=T})\tilde{\phi} d\Omega \tag{3.12}$$

$$- \int_0^T \int_\Omega \Lambda(\frac{\partial \tilde{\phi}}{\partial t} + \beta\nabla\tilde{\phi}) d\Omega dt - \int_0^T \int_\Gamma \tau(\tilde{\phi} - \tilde{g}_D) d\Gamma dt \tag{3.13}$$

$$- \int_\Omega \eta(\tilde{\phi} \mid_{t=0}) d\Omega \tag{3.14}$$

Applying integration by parts derives the adjoint equation. We obtain

$$\frac{\partial L}{\partial \phi} = \int_0^T \int_\Omega \tilde{\phi}\Big(\frac{\partial \Lambda}{\partial t} + \beta\nabla\Lambda) + \gamma_1(\phi - \Phi)\Big) d\Omega dt \tag{3.15}$$

$$- \int_\Omega (\Lambda \mid_{t=T} - \gamma_2(\phi \mid_{t=T} - \Phi \mid_{t=T}))\tilde{\phi} \mid_{t=T} d\Omega \tag{3.16}$$

$$- \int_0^T \int_\Gamma \tau(\tilde{\phi} - \tilde{g}_D) d\Gamma dt + \int_\Omega \tilde{\phi} \mid_{t=0} (\Lambda \mid_{t=0} - \eta) d\Omega \tag{3.17}$$

$$- \int_0^T \int_\Omega \tilde{\phi} \cdot \Lambda d\Gamma dt \tag{3.18}$$

The two equations which result from the Lagrangian multipliers τ and η are not really relevant for the adjoint problem. Adjoint system given then is

$$-\frac{\partial \Lambda}{\partial t} - \beta\nabla\Lambda = \gamma_1(\phi - \Phi) \quad (T, 0) \times \Omega \tag{3.19a}$$

$$\Lambda = 0 \quad (T, 0) \times \Gamma \tag{3.19b}$$

$$\Lambda = \gamma_2(\phi \mid_{t=T} - \Phi \mid_{t=T}) \quad \Omega \text{ at } t = T. \tag{3.19c}$$

is also an advection equation but with the computation time reversed. The source is generated by the first integral term of the time dependent time cost function J in (3.1). The initial condition of the adjoint equation is generated by the second integral of the time dependent time cost function J in (3.1).

Setting the derivative of L w.r.t. to the optimization parameter equal to zero, we could derive the so called optimality conditions. The "one-shot method" combines these with the PDE and the adjoint equation to form the optimality system. According to [1], advantage of the one-shot method, is that if the combined system is solvable, the solution is also directly the desired optimum. However, as the number of unknowns in the combined system is more than double that of the PDE system, solving the optimality system directly is often not possible. Even iterative methods may not converge.

A perhaps more intuitive way to compute the derivative is shown in [1] by go via the sensitivity equation. The sensitivity equation can be obtained by differentiating the PDE equation with w.r.t. to the optimization parameter β using the chain rule:

$$\frac{\partial}{\partial \beta}\Big(\frac{\partial \phi}{\partial t} + \beta(\nabla \cdot \phi) = 0\Big) = 0. \tag{3.20}$$

To simplify the partial derivative notation, the partial w.r.t. to the parameter is denoted as: $\phi_\beta = \frac{\partial \phi}{\partial \beta}$. The sensitivity equation for the advection is:

$$\begin{aligned} \frac{\partial \phi_\beta}{\partial t} + \beta \nabla \cdot \phi_\beta &= -\nabla \cdot \phi \quad (0,T) \times \Omega \\ \phi_\beta &= 0 \quad (0,T) \times \Gamma \\ \phi &= 0 \quad \text{in } \Omega \text{ at } t = 0 \end{aligned} \tag{3.21a}$$

For a PDE with K parameters, this would result in K sensitivity equations. To compute the total derivative for each parameter, K equations must be solved. Already for a small number of $3-5$ parameters this would be costly. The reason for making a larger effort in further deriving the adjoint equation (3.19a), is that it must only be set up once and can be used to compute the derivative for multiple parameters. Instead of solving K equations, only a single equation is solved and then used to compute the K different derivatives.

3.2.2. Deriving the adjoint system using the product rule

A more general formulation of the adjoint and gradient derivation is given in [1] and [2]. In this section we follow this general approach, to obtain a matrix-vector formulation of the gradient. The general formulation of the cost function

$$J(\alpha, \phi(\alpha))$$

contains a direct dependency on the parameter α and the state variable ϕ but also has an indirect dependency on α via ϕ. We follow the derivation steps in [1] and compute the gradient of the cost functional by applying the chain rule to obtain the total derivative:

$$\frac{DJ}{D\alpha} = \frac{\partial J}{\partial \phi}\frac{d\phi}{d\alpha} + \frac{\partial J}{\partial \alpha} \tag{3.22}$$

The explicit formulation of the cost functional J is known, whether this is in exact analytical form or as a discrete function. The terms $\frac{\partial J}{\partial \phi}$ and $\frac{\partial J}{\partial \alpha}$ can therefore be computed normally using the standard derivation rules. On the other side, the Jacobian matrix $\frac{d\phi}{d\alpha}$ of the state variable ϕ w.r.t. the parameter α is not trivial to compute. In most cases there is no explicit formulation of the PDE given, which we could directly differentiate. Additionally, the derivative matrix is fully dense and therefore expensive to compute and store [2]. In order to avoid the direct computation of this term, the adjoint equation

$$\Big(\frac{\partial F}{\partial \phi}\Big)^T \Lambda = \Big(\frac{\partial J}{\partial \phi}\Big)^T, \tag{3.23}$$

given in [1] can be substituted into (3.22) giving:

$$\frac{DJ}{D\alpha} = \Lambda^T \frac{\partial F}{\partial \phi}\frac{d\phi}{d\alpha} + \frac{\partial J}{\partial \alpha}. \tag{3.24}$$

The adjoint equation (3.23) was derived in [1] by setting up the Lagrangian function, as show in section (3.2.1), but for the more general formulation. Finally, the term $\frac{\partial F}{\partial \phi}\frac{d\phi}{d\alpha}$ can be replaced by inserting the sensitivity equation. We can derived it by applying the chain rule to the PDE system w.r.t. state variables as

$$\begin{aligned} \frac{d}{d\alpha}F(\phi,\alpha) &= 0 \\ \frac{\partial F}{\partial \phi}\frac{d\phi}{d\alpha} + \frac{\partial F}{\partial \alpha} &= 0 \\ \frac{\partial F}{\partial \phi}\frac{d\phi}{d\alpha} &= -\frac{\partial F}{\partial \alpha}. \end{aligned} \tag{3.25}$$

Inserting (3.25) into (3.24) gives the equation

$$\frac{DJ}{D\alpha} = -\Lambda^T \frac{\partial F}{\partial \alpha} + \frac{\partial J}{\partial \alpha} \tag{3.26}$$

for computing the derivative of the functional w.r.t. the parameters. Slightly different derivation steps are shown in [2], which derive the adjoint equation by going via the sensitivity equation. If we inverting the last line of the sensitivity equation, we obtain

$$\frac{d\phi}{d\alpha} = \Big(\frac{\partial F}{\partial \phi}\Big)^{-1}\Big(-\frac{\partial F}{\partial \alpha}\Big). \tag{3.27}$$

This equation can then be used to replace the problematic derivative $\frac{d\phi}{d\alpha}$ in (3.22) giving

$$\frac{DJ}{D\alpha} = \frac{\partial J}{\partial \phi}\Big(\frac{\partial F}{\partial \phi}\Big)^{-1}\Big(-\frac{\partial F}{\partial \alpha}\Big) + \frac{\partial J}{\partial \alpha}. \tag{3.28}$$

By taking the Hermitian transpose of the system

$$\frac{DJ^*}{D\alpha} = \frac{\partial J^*}{\partial \phi} \underbrace{\Big(\frac{\partial F}{\partial \phi}\Big)^{-*}\Big(-\frac{\partial F}{\partial \alpha}\Big)^{*}}_{=\Lambda} + \frac{\partial J^*}{\partial \alpha}. \tag{3.29}$$

one can obtain the same adjoint equation and total derivative as in Equation (3.26). As we can assume that the derivative values are real valued, we replace the Hermitian A^* with the transpose A^T when using the formulations given in the following sections.

3.3. Gradient computation using the adjoint

3.3.1. Gradient formulation using the Lagrangian: Advection equation

Now that the adjoint equation was obtained via the Lagrangian, we can compute the total derivative by differentiating the cost function with respect to the optimization parameters, in our example this is the flow velocity. By deriving J w.r.t. β we obtain:

$$\frac{DJ}{D\beta} = \int_0^T \int_\Omega (\phi - \Phi)\phi_\beta d\Omega dt + \int_\Omega \Big(\phi \mid_{t=T} - \Phi \mid_{t=T}\Big)\phi_\beta \mid_{t=T} d\Omega \tag{3.30}$$

$$= \int_0^T \int_\Omega \Big(-\frac{\partial \Lambda}{\partial t} - \beta \nabla \cdot \Lambda\Big)\phi_\beta d\Omega dt + \int_\Omega (\Lambda)\phi_\beta \mid_{t=T} d\Omega \tag{3.31}$$

Now we can use the adjoint equation to replace the term $\phi - \Phi$ and the terminal condition of the adjoint equation to replace the term $\phi \mid_{t=T} - \Phi \mid_{t=T}$. Integrating the resulting system by parts over time and space gives:

$$\frac{DJ}{D\beta} = \int_\Omega \int_0^T \Lambda \frac{\partial \phi_\beta}{\partial t} dt d\Omega - \int_\Omega (\Lambda \phi_\beta) \mid_{t=0}^{t=T} d\Omega \tag{3.32}$$

$$+ \int_0^T \int_\Omega \Lambda \nabla \cdot \phi_\beta d\Omega dt - \int_0^T (\phi_\beta \nabla \cdot \Lambda \cdot \vec{n}) \mid_\Gamma dt \tag{3.33}$$

$$+ \int_\Omega \Lambda \phi_\beta \mid_{t=T} d\Omega \tag{3.34}$$

By re-arranging the equation system

$$\frac{DJ}{D\beta} = \int_\Omega \int_0^T \Lambda \Big(\frac{\partial \phi_\beta}{\partial t} + \nabla \cdot \phi_\beta\Big) dt d\Omega - \int_\Omega (\Lambda \phi_\beta) \mid_{t=0}^{t=T} d\Omega \tag{3.35}$$

$$- \int_0^T (\phi_\beta \nabla \cdot \Lambda \cdot \vec{n}) \mid_\Gamma dt \tag{3.36}$$

$$+ \int_\Omega \Lambda \phi_\beta \mid_{t=T} d\Omega, \tag{3.37}$$

we can see how the conditions from (3.21a) and (3.19a) cancel out all single integral terms or are equal to zero. The equation with ϕ_β can be substituted with the right hand side of the sensitivity equation. The total derivative can therefore be expresses as:

$$\frac{DJ}{D\beta} = \int_\Omega \int_0^T \Lambda \cdot \phi_\beta \; dt d\Omega \tag{3.38}$$

Gradient formulation using the Lagrangian: Elastic wave equation

For the second test equation, the elastic wave equation, we have to do the same steps for deriving the adjoint equation and total derivative formulation. We again follow the derivation steps given in [1]. In the following we only make a first attempt at the derivation of the adjoint equation. We follow the first steps to show that we expect the elastic wave adjoint to be self-adjoint [11].

The general optimal control problem is:

$$\begin{aligned} &\underset{C}{\text{minimize}} \quad J(u, C, a, b, c) \\ &\text{subject to} \quad F(u, C, a, b, c) = \rho \frac{\partial^2 u}{\partial t^2} - \nabla \cdot \sigma = 0 \quad \text{in } \Omega \times (0, T) \\ &\qquad\qquad \sigma \cdot n = T \quad \text{in } \Gamma \times (0, T) \\ &\qquad\qquad u = u_0(x) \quad \text{in } \Omega \text{, t=0} \end{aligned}$$

Setting up the Lagrangian the same way as shown previously, we obtain

$$\begin{aligned} L(u, C, a, b, d) = J(u, C) &- \underbrace{\int_0^T \int_\Omega a\Big(\rho \frac{\partial^2 u}{\partial t^2} - \nabla\sigma\Big) \, d\Omega dt}_{F(u,C,a)} \\ &- \underbrace{\int_0^1 \int_\Omega b \cdot (\sigma \cdot n - T) d\Gamma dt}_{BC(u,C,b)} \\ &- \underbrace{\int_\Omega c \cdot (u \mid_{t=0} - u_0(x)) \; d\Omega}_{IC(u,C,c)} \end{aligned} \tag{3.39}$$

Deriving w.r.t. the Lagrangian multipliers a

$$\begin{aligned} \frac{\partial L}{\partial a} &= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \Big(- F(u, C, a + \epsilon \tilde{a}) - (-F(u, C, a)) \Big) \\ &= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \Big(\int_0^T \int_\Omega \epsilon \tilde{a} (\rho \frac{\partial^2 u}{\partial t^2} - \nabla\sigma) \; d\Omega dt \Big) \\ &= \int_0^T \int_\Omega (\rho \frac{\partial^2 u}{\partial t^2} - \nabla\sigma) \; d\Omega dt \end{aligned} \tag{3.40}$$

will make $\frac{1}{\epsilon}$ and ϵ cancel out and we obtain the original PDE system again, as we can arbitrarily chose the variation. This can be done the same way for the other two multipliers. To obtain the adjoint equation system, we derive w.r.t. the state variable u.

$$\begin{aligned}\frac{\partial L}{\partial u} &= \lim_{\epsilon \to 0} \frac{1}{\epsilon}\Big(L(u+\epsilon\tilde{u}) - L(u)\Big) \\ &= \lim_{\epsilon \to 0} \frac{1}{\epsilon}\Big(J(u+\epsilon\tilde{u}) - J(u) \\ &\qquad - F(u+\epsilon\tilde{u}) + F(u) \\ &\qquad - BC(u+\epsilon\tilde{u}) + BC(u) \\ &\qquad - IC(u+\epsilon\tilde{u})) + IC(u)\Big)\end{aligned} \tag{3.41}$$

Two relations given in [11] between the material parameter matrix C, the stress σ and the displacement u are used to rearrange the equations.

$$\sigma + \epsilon\tilde{\sigma} = C : E[u+\epsilon\tilde{u}] = C : E[u] + C : E[\epsilon\tilde{u}] \tag{3.42}$$

Following [11], the divergence of the stress $\nabla\sigma$ can be rewritten as:

$$\nabla \cdot \sigma + \epsilon\nabla \cdot \tilde{\sigma} = \nabla C : \nabla(u+\epsilon\tilde{u}) = \nabla C : \nabla u + \nabla C : \nabla\epsilon\tilde{u} \tag{3.43}$$

For the F term the derivation w.r.t. the state variable can be determined as:

$$= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \int_0^T \int_\Omega a\Big(-\rho\frac{\partial^2 u + \epsilon\tilde{u}}{\partial t^2} + \nabla(\sigma + \epsilon\tilde{\sigma}) + \rho\frac{\partial^2 u}{\partial t^2} - \nabla\sigma\Big) d\Omega dt \tag{3.44}$$

$$= \int_0^T \int_\Omega a\Big(-\rho\frac{\partial^2 \tilde{u}}{\partial t^2} + \nabla\tilde{\sigma}\Big) d\Omega dt \tag{3.45}$$

Integrating by parts twice over time to move derivative onto Lagrangian multiplier gives us:

$$\int_0^T -\rho a \frac{\partial^2 \tilde{u}}{\partial t^2} dt = \int_0^T -\rho\tilde{u}\frac{\partial^2 a}{\partial t^2} dt + \Big[\rho\tilde{u}\frac{\partial^2 a}{\partial t^2}\Big]_0^T + \Big[\rho a\frac{\partial \tilde{u}}{\partial t}\Big]_0^T \tag{3.46}$$

Then we integrate by parts over space once and obtain

$$\int_\Omega \nabla\tilde{\sigma} dt = \int_\Omega (\nabla C) : (\nabla\tilde{u}) dt = -\int_\Omega (\nabla \cdot C) : (\nabla a) \cdot \tilde{u} d\Omega \tag{3.47}$$

$$+ \int_\Gamma a(\nabla \cdot C) : (\tilde{u}) d\Gamma \tag{3.48}$$

If we define $(\nabla \cdot C) : (\nabla a) = \nabla \cdot \sigma_a$, we can see the original elastic wave equation but now with the adjoint variable a as the unknown:

$$= \int_0^T \int_\Omega \tilde{u}\Big(-\rho\frac{\partial^2 a}{\partial t^2} - \nabla \cdot \sigma_a\Big) dt d\Omega \tag{3.49}$$

$$+ \Big[\rho\tilde{u}\frac{\partial^2 a}{\partial t^2}\Big]_0^T + \Big[\rho a\frac{\partial \tilde{u}}{\partial t}\Big]_0^T + \int_\Gamma a(\nabla \cdot C) : (\tilde{u}) d\Gamma \tag{3.50}$$

We know from [11], that the elastic wave equation should be self-adjoint, meaning that the adjoint equation is again an elastic wave equation. We can see this also in our derivation attempt for the adjoint equation in (3.50). The remaining steps would include applying the derivation w.r.t. to the state variable ϕ to the boundary and initial condition parts of the Lagrangian.

3.4. Adjoint system for time dependent PDEs

The above derivations both did not show directly the time dependency of the adjoint equation. In this section we closely follow the steps in [15] for computing the gradient using the adjoint solutions over the complete time. For a time dependent system the adjoint equation is actually a system of equations

$$\underbrace{\left[\quad \frac{\partial F}{\partial \phi} \quad\right]^T}_{(n^2\cdot m)\times(n^2\cdot m)} \underbrace{\vec{\Lambda}}_{(m\cdot n^2)\times 1} = \underbrace{\left(\frac{\partial \vec{J}}{\partial \phi}\right)^T}_{(1\times m\cdot n^2)} \tag{3.51}$$

with a row per time level of the forward problem. The dimensions of the partial derivatives depend on the dimensions of the given optimization problem. The example considered in [15], computes the adjoint equation for the one dimensional advection equation using Finite Differences to solve the PDE equation. The main steps in [15] are however very general and can easily be used for our general example of a two dimensional PDE $F = 0$ with the state vector ϕ which has v variables. The domain is discretized on a $(n \times n)$ grid, with the indexes i, j = {(1,1),(2,1), …, (n,n)}. The time is discretized with $k = 0, \ldots, m$ time steps. All time levels of the system variable ϕ are stored in the vector $\vec{\phi} = [\phi^1_{11}, \phi^1_{21}, \ldots, \phi^1_{n1}, \ldots, \phi^m_{nn}]^T$ of size $(m \cdot n^2 \times 1)$. The right hand side of the equation system is given by the derivative of the cost function with respect to the state variable. As the cost function J is a single value per variable v, the derivative $\frac{\partial J}{\partial \alpha}$ is also a scalar. The partial derivative $\frac{\partial J}{\partial \phi}$ is a vector of size $(1 \times m \cdot n^2)$. The adjoint vector, the unknowns, is $\Lambda = [\Lambda^0_{11}, \ldots, \Lambda^0_{1n}, \ldots, \Lambda^{m-1}, \Lambda^m]^T$ of the size $(m \cdot n^2 \times 1)$. To remind ourselves, the adjoint equation is solved after having computed the forward solution ϕ.

Computing the derivative matrix $\frac{\partial F}{\partial \phi}$

We start by looking at setting up the system matrix of the adjoint equation (3.51). If the complete matrix $\frac{\partial F}{\partial \phi}$ for all time levels would be set up, it would be a $(m_B \times m_B)$ block matrix with blocks of size $(n^2 \times n^2)$ which are the partial derivatives in space of a single time step. As shown in the example given in [15], only the diagonal and sub-diagonals contain non-zero matrix blocks as shown in (3.52). In the matrix the rows represent the time levels. The PDE equation F^k at time step k is only dependent on solutions ϕ^l where

$l \leq k$. If the derivative of F^k w.r.t. to a solution vector ϕ^l at a time step $l > k$ is made, this would clearly be zero. This means all derivatives above the diagonal are zero.

$$\frac{\partial F}{\partial \phi} = \begin{bmatrix} \frac{\partial F^0}{\partial \phi^0} & 0 & \dots & 0 \\ \frac{\partial F^1}{\partial \phi^0} & \frac{\partial F^1}{\partial \phi^1} & \dots & 0 \\ \vdots & & & \vdots \\ \frac{\partial F^m}{\partial \phi^0} & \frac{\partial F^m}{\partial \phi^1} & \dots & \frac{\partial F^m}{\partial \phi^m} \end{bmatrix} \tag{3.52}$$

In practice, setting up the large time dependent system is not necessary, as the matrix is mostly sparse depending on the chosen time stepping. We use the same time stepping method as in [15], the Euler method method

$$F^k = \frac{1}{\Delta t}\left(\phi^k - \phi^{k-1}\right) + \Delta\phi = 0 \tag{3.53}$$

but using the backward definition [16]. Each time level k depends on the previous level $(k-1)$. The resulting derivative matrix would only have non-zero block entries on the diagonal and the sub-diagonal as shown in (3.57).

Each entry of the full time matrix is in turn a derivative matrix but in x and y-direction [15]. In each matrix row contains two non-zero partial derivative matrices at the time step k and $(k-1)$. The partial derivatives of F at the time step k can easily be determined by deriving the Euler equation w.r.t. ϕ^k, as a diagonal matrix (3.54) with the time step $\frac{1}{\Delta t}$ on the diagonal.

$$U_1 = \frac{\partial F^k_{ij}}{\partial \phi^k_{ij}} = \begin{bmatrix} \frac{1}{\Delta t} & & \\ & \ddots & \\ & & \frac{1}{\Delta t} \end{bmatrix} \tag{3.54}$$

The sub-matrix (3.55) of the partial w.r.t. to the state vector ϕ^k at time step $(k-1)$ is also a sparse matrix, with non-zero entries depending on the chosen discretization scheme in space [15]. The matrix is given in the transposed form as:

$$U_2 = \left(\frac{\partial F^k_{ij}}{\partial \phi^{k-1}_{ij}}\right)^T = \begin{bmatrix} \frac{\partial F^k_{i,1}}{\partial u^{k-1}_{i,1}} & \frac{\partial F^k_{i,2}}{\partial u^{k-1}_{i,1}} & 0 & 0 & \dots & 0 \\ 0 & \frac{\partial F^k_{i,2}}{\partial u^{k-1}_{i,2}} & \frac{\partial F^k_{i,3}}{\partial u^{k-1}_{i,2}} & 0 & \dots & 0 \\ 0 & 0 & 0 & & & \\ & & & & \ddots & \frac{\partial F^k_{i,n}}{\partial u^{k-1}_{i,n-1}} \\ & & & & & \frac{\partial F^k_{i,n}}{\partial u^{k-1}_{i,n}} \end{bmatrix} \tag{3.55}$$

As the equation system is two dimensional, every matrix entry is the y-derivative at j, for every point i in x-direction. The structure of theses sub-sub-derivative matrices stays the same. The matrix example above is given disregarding boundary conditions. For a first order Finite Volume (FV) stencil, approximating the flux with the local Lax-Friedrichs Method, each row contains at most five non-zero entries. Matrix $\bar{U}_2$ in (3.56) gives real valued example for a small 9×9 grid when applying a FV discretization on the two dimensional advection equation. Here we can also see how the boundary conditions changes the sparsity pattern of the derivative matrix.

$$\bar{U}_2 = -\begin{bmatrix} 1.5 & 0.0 & 0.75 & 0.0 & 0.0 & 0.0 & 0.75 & 0.0 & 0.0 \\ 0.75 & 1.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.75 & 0.0 \\ 0.0 & 0.75 & 1.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.75 \\ 0.75 & 0.0 & 0.0 & 1.5 & 0.0 & 0.75 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.75 & 0.0 & 0.75 & 1.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.75 & 0.0 & 0.75 & 1.5 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.75 & 0.0 & 0.0 & 1.5 & 0.0 & 0.75 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.75 & 0.0 & 0.75 & 1.5 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.75 & 0.0 & 0.75 & 1.5 \end{bmatrix} \tag{3.56}$$

Iterative scheme for the adjoint computation

For the analytical adjoint of the advection equation, it was shown that the time and advection direction is reversed [2].

$$\begin{bmatrix} U_1^1 & U_2^2 & 0 & 0 & 0 \\ 0 & U_1^2 & U_2^3 & 0 & 0 \\ & & \ddots & \ddots & \\ 0 & 0 & 0 & U_1^{m-1} & U_2^m \\ 0 & 0 & 0 & 0 & U_1^m \end{bmatrix} \cdot \begin{bmatrix} \Lambda^0 \\ \Lambda^1 \\ \vdots \\ \Lambda^{m-1} \\ \Lambda^m \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial \phi^0} \\ \frac{\partial J}{\partial \phi^1} \\ \vdots \\ \frac{\partial J}{\partial \phi^{m-1}} \\ \frac{\partial J}{\partial \phi^m} \end{bmatrix} \tag{3.57}$$

We now can see the same: Starting with a terminal condition the end time T with Λ^m, the equation system (3.57) is solved using back substitution for the initial solution Λ^0. For a PDE, for which the right hand side is independent of time, each time step will have the same derivative. This means the time dependent system of equations (3.57) simplifies further, as $U_1^k = U_1^{k+1}$ and $U_2^k = U_2^{k-1}$. The iterative scheme given in [15] then reads as:

1. Start with $U_1 \Lambda^m = \frac{\partial J}{\partial \phi^m}$
2. For $k = m-1, \dots, 1$

$$U_1 \lambda^k = \Big(\frac{\partial J}{\partial \phi^k} - \lambda^{k+1} U_2\Big) \tag{3.58}$$

Part III.

Implementation

4. Implementation

4.1. FEniCS and `Dolfin-Adjoint` using Python

FEniCS is an open-source finite-element solver which translates a weak form written using a python interface in UFL (Unified Form Language), into C++ Finite Element code [4], [5], [17], [18], [19], [20]. The major advantage of this is that the UFL problem formulation is very similar to the mathematical formulation. The package dolfin-adjoint overloads certain FEniCS methods, to allow the computation of the gradient using the adjoint. We use FEniCS and dolfin-adjoint in the next section to implement the forward models of the test equations and setup the optimal control problems.

4.1.1. Basic methods and data structures

We shortly go over the basic methods in FEniCS , as explained in the online tutorial book [21]. In FEniCS every function is generated over a function space, which is generated from the given domain mesh. The basic function space is generated by using the method `FunctionSpace`. If every unknown in the function space is to be a vector, then one has to set up a `VectorFunctionSpace` with the given vector dimension `dim=N`. The type of basis element is set by the parameter `family`, e.g. for standard continuous Lagrangian basis functions the keyword `CG` is passed. For Discontinuous Galerkin elements the keyword `DG` is together with the element degree. The function `Measure` returns an operator which computes the integral over a certain domain in the mesh. The integral over the domain area is given by `dx`, while `dS` gives the integral over the inner facets and `ds` the integral over the boundary facets.The method `FacetNormal` returns the outward pointing face normal of the given mesh. Examples for these basic methods can be seen in the code extract (4.1). The FEniCS Expression class allows the user to define an C++ expression which then is JIT (just-in-time) compiled, avoiding the overhead when making calls to a python defined function from the auto-generated C++ code [21]. An example is given for the Lamé parameter definition in code listing (4.7). FEniCS also provides a simple-to-use interface for imposing strong Dirichlet and von-Neumann boundary conditions. The method `DirichletBC(V, gD, on_boundary)` takes the function space, an expression `gD` which gives the values at the boundary and a user defined function `on_function` that returns a Boolean value if the mesh point x is on the boundary or not [21]. A very simple example is `on_boundary = lambda x : near(x[0], 0, tol)`, which only returns true if the point x is on the left boundary. When the `solve` function is called, the boundary object `bcs` is passed to be applied to the system matrices.

```
V_dg = FunctionSpace(unit_mesh, 'DG', dg_order)
V_vec_dg = VectorFunctionSpace(unit_mesh, 'DG', dg_order, dim=5)

u, v = TrialFunction(V), TestFunction(V)

dx = Measure('dx', domain=unit_mesh)   # domain area
dS = Measure('dS', domain=unit_mesh)   # interior facet
ds = Measure('ds', domain=unit_mesh)   # boundary facets

n = FacetNormal(unit_mesh) # get the normal vector
```

Source Code 4.1.: For a basic FEniCS setup a function space using a specified FE-element type is defined. The corresponding trial and test functions can easily be obtained from `V`. The measure objects `dx, dS, ds` compute the integral over a certain subdomain. The face normal $\vec{n}$ is the outward pointing normal vector on the cell facet.

4.1.2. Weak formulation for the hyperbolic wave equation using DG-elements

In FEniCS the general variational formulation is

$$a(u, v) = L(v), \tag{4.1}$$

where the left hand side a is dependent on both the u, the unknown state variable and the test function v [21]. All known terms are collected in the right hand side $L(v)$. An equivalent formulation is given by

$$F(u, v) = 0, \tag{4.2}$$

if the sides are not split. The DG formulation defined in (2.2) is generated with the method `L_operator` as shown in the code extract (4.2). The main difference when using Discontinuous Galerkin FE-elements is that the boundary conditions must be imposed weakly in the variational formulation as defined in equation (2.2). Translated into ufl code, source code extract (4.2) shows a general method for generating the weak formulation. As shown in [22], the facet normal must be restricted when integrating over the interior facets. This can be done using the restriction operator as shown in [23], which selects ether the right `'+'` or the left `'-'` side of the facet. The physical flux and the maximum velocity, i.e. the maximum eigenvalue are equation specific. For the advection equation the flux passed would simply be `flux = beta * phi_old` as shown in [24] and for the elastic wave equation the flux definition is given in 4.5. The dimensions of the flux must match the dimensions and number of variables of the function space `V`, e.g. for the elastic wave equation the `flux.ufl_shape` is `(5,2)`. The general structure of a forward solve method using an explicit Euler time discretization is shown in (4.3). To add a higher order time stepping, we follow the example given in [25], which shows how to implement the stabilized third

```python
def L_operator(phi_old, flux, maxvel, gD, v, f, n):
    L = dot(grad(v), flux)*dx + dot(f, v)*dx
    flux_dS = dot(avg(flux), n('+')) + 0.5*maxvel('+')*jump(phi_old)
    flux_ds = 0.5*(dot((flux + gD*beta), n) + maxvel*(phi_old - gD))
    L -= inner(flux_dS, jump(v)) * dS
    L -= inner(flux_ds, v) * ds
    return L
```

Source Code 4.2.: UFL implementation of the variational formulation for the hyperbolic system F as defined under (2.1). The face normal in the interior integral `flux_dS` must be restricted as shown in [22]. The boundary is weakly imposed by using the boundary function g_D to define flux at the boundary.

```python
def forward(...):
    ... # Basic setup of Function Space V ect.
    u1 = Function(V, name='u1')
    u1 = project(ic, V) # project IC
    # variational formulation and time discretization
    F = - inner(dt * (u - u1), v) * dx # Euler time stepping
    F += L_operator(beta, u, gD, v, f, n)
    a, L = lhs(F), rhs(F)

    u_new = Function(V) # storage for computed time step n + 1
    steps = int(t_end / dt) # set up time stepping
    t = dt
    gD.t = dt

    for n in range(1, steps):
        solve(a == L, u_new) # Alternatively: solve(F == 0, u_new)
        u1.assign(u_new) # update the variable
        t += delta_t
        gD.t = t # update boundary expression

    return u1, t
```

Source Code 4.3.: FEniCS implementation of the advection equation using DG elements and Euler time stepping.

order strong-stability-preserving Runge-Kutta time stepping scheme given in [26]. For this the variational formulation is setup without the time discretization term. The code extract (4.4) shows how the right hand side for an s-stage RK scheme is set up s times and the left hand side is defined as the inner product of u and v [25]. The solutions for the different time levels are then combined together correctly inside the time loop [25].

```
a = inner(v, u) * dx
u1 = project(gD, V)
u2 = Function(V) # create function to hold intermediate result
L1 = L_operator(beta, u1, gD, v, f, face_normal)
L2 = L_operator(beta, u2, gD, v, f, face_normal)

# RK2 uses two intermediate time steps
for n in range(1, steps):
    solve(a == L1, u_new)
    u2.assign(u1 + delta_t * u_new)
    gD.t = t + delta_t
    solve(a == L2, u_new)
    u1.assign(0.5 * (u1 + u2 + delta_t * u_new))
```

Source Code 4.4.: Higher order time-stepping using the Runge-Kutta order 2 midpoint scheme.

For the advection equation, defining the variational formulation in ufl is quite simple. However, for more complex systems the implementation can become quite complex. This makes it interesting to work with the package `dofin_dg` which provides structured utility functions for automatically generating the DGFEM variational formulation for more complex systems [9]. The method `L_operator` can instead be implemented using the *dolfin_dg* class `HyperbolicOpterator`. The hyperbolic operator provides a structured base class for setting up the weak formulation. The flux computation is defined inside a flux operator e.g. the `LocalLaxFriedrichs`. The package however throws two compile errors if directly used with dolfin-adjoint. To make the package compatible, the interior flux is slightly re-written. The eigenvalue must also be restricted and the average function `avg` is used to compute the restriction. Also, as we can define an explicit expression for the eigenvalues in our test example, the max eigenvalue `alpha` is defined as a ufl-expression.

4.1.3. Computation of the gradient using Dolfin-Adjoint and parameter optimization

Now that we have shown how to implement a FEniCS model, we look at how dolfin-adjoint computes the gradient for the optimal control problem. As explained in [6], the idea of dolfin-adjoint is to overload a FEniCS module function with a `pyadjoint.Block` subclass which can compute the value and also the derivative. The basic modules such as `Function`, `solve` and `assign` are automatically overloaded. The gradient values are recorded on a tape, which can be turned on and off by setting the parameter `annotate` to true or false [7]. During the initial forward solve, dolfin-adjoint creates a dependency graph of all blocks [6]. If a block object is connected to the graph, it is required for the adjoint computation and dolfin-adjoint will automatically record it in the background during the optimization solve.

```python
class LocalLaxFriedrichsModified(ConvectiveFlux):
    def interior(self, F_c, u, n):
        return dot(avg(F_c(u)), n) + 0.5 * self.alpha('+') * jump(u)
    def exterior(self, F_c, u_p, u_m, n):
        return 0.5 * (dot(F_c(u_p), n) + dot(F_c(u_m), n) /
                + self.alpha * (u_p - u_m))

# operator derived from dolfin_dg base class
class ElasticOperatorLOH1(HyperbolicOperatorModified):
    ...
    def F_c(self, U):
        sig11, sig22, sig12, u, v = U[0], U[1], U[2], U[3], U[4]
        return as_matrix([[-(self.lam+2*self.mu)*u,-self.lam*v],
                          [-self.lam*u,-(self.lam + 2*self.mu) * v],
                          [-self.mu*v,-self.mu*u],
                          [-(sig11/self.rho), -(sig12/self.rho)],
                          [-(sig12/self.rho), -(sig22/self.rho)]])

    def __init__(self, mesh_, V, bcs, parameters):
        self.parameters = parameters
        self.domain_mesh = mesh_
        self.lam, self.mu, self.rho = get_expressions(parameters)

        HyperbolicOperatorModified.__init__(self, mesh_, V, bcs,
        self.F_c, LocalLaxFriedrichsModified(parameters))
...
elasticOp = LOH1Operator(unit_mesh, V, DGDirichletBC(ds, gD),parameters)
F = elasticOp.generate_fem_formulation(u, v) # auto-generated DGFEM
F += dot((1 / dt) * (u - u1), v) * dx # add time discretization
F -= inner(f, v) * dx
```

Source Code 4.5.: The LOH1Operator is derived from the base class HyperbolicOperator which is provided by `dolfin_dg` [9]. The classes `HyperbolicOperator` and `LocalLaxFriedrichs` are only slightly modified to use a user defined expression as maximum eigenvalue, which is also restricted for the inner flux approximation.

In order to compute the derivative of the cost function J, some modifications to the FEniCS code have to be made. In the following section we use the basic setup for a parameter optimization problem given in [7]. The code listing (4.6) show the basic setup for a parameter optimization. The dolfin-adjoint object `Control` is used to define the optimization parameter. Then the forward solve method has to be run once, so that dolfin-adjoint can record the annotated operations. The cost function J can be defined using the assemble method. If the cost function includes the integral over time, the evaluation over the domain has to be done inside the forward solve at every time step, recorded in a python list which can after the time stepping be integrated over time [7]. The minimize method calls the `scipy` optimization routines e.g. the standard is the quasi newton method L-BFGS-B [27]. For a user-defined expression the changes can not automatically be traced and dolfin-adjoint requires the list of dependencies together with the user defined derivatives of the expression with respect to each dependency. We use the same solution approach for such a case given in the documentation example [28]. For example, the Lamé parameter ρ of the LOH1 problem is formulated as a user-expression using a switch case as the value of the density changes depending on x being in the layer or in the half space. This means the expression is dependent on the two FEniCS constants `rho_layer` and `rho_hs` (half-space). As the derivative with respect to the layer depth is not defined, the dependency is not added but also cannot be used as a control variable. A solution would be to approximate the step function with a smooth function to ensure that the derivative w.r.t. the layer depth is well defined.

```
control = Control(rho_layer) # optimization parameter
u_sol = forward(..) # initial run of forward model to record adjoint

J = assemble(inner(u_sol, u_sol) * dx) # Cost function definition
J_hat = ReducedFunctional(J, control)
# interface to scipy optimization routine
opt_res = minimize(J_hat, method="L-BFGS-B", ftol=1.0e-8,
options=...)
```

Source Code 4.6.: Basic dolfin-adjoint set up for a parameter optimization [7].

The code for the parameter optimization in listing (4.6), makes the following steps:

1. Define a control variable for the optimization parameter
2. Run the forward model once so that dolfin-adjoint can create a dependency graph
3. Define the cost function J as the squared difference between the initial forward solution and the measurements
4. Compute the derivative $\hat{J}$ with respect to the optimization parameter variable
5. Determine the reduced functional $\hat{J}$
6. Start the optimization routine which will use the forward model to compute the cost function J in every step and its derivative

```
rho = Expression("x[1] >= divide ? rho_layer : rho_hs",
                degree=1, name='rho_exp', divide=divide,
                rho_layer=rho_layer, rho_hs=rho_hs)

rho.dependencies = [rho_layer, rho_hs]  # add dependencies and derivatives
d1 = Expression("x[1] >= divide ? 1.0 : 0.0", divide=divide, degree=1)
d2 = Expression(...)
rho.user_defined_derivatives = {rho_layer: d1, rho_hs: d2}
```

Source Code 4.7.: Dolfin-adjoint modifications: providing the dependency list with the corresponding user defined derivatives for an expression [21].

4.2. TerraDG using Julia

For a comparison, we extend the discontinuous Galerkin solver TerraDG, which was provided by [3] with an adjoint solver. For comparison we implement two different approaches for computing the total derivative. A short overview of both approaches is given in the following:

1. *differentiate-first-then-discretize*:

   The adjoint equation is derived analytically via the Lagrangian approach. The forward equation and the adjoint equation are then discretized using the Discontinuous Galerkin method. The discrete equations are then solved using TerraDG [3]. The total derivative is computed by the integral formulation also derived analytically using the Lagrangian function.

2. *discretize-first-then-differentiate*:

   The cost function J and the PDE model F are first discretized using the Discontinuous Galerkin method. The DG-solver TerraDG is used to solve the forward model and compute the cost function. The total derivative is then computed by applying automatic-differentiation (AD) to the solver methods.

The advection equation is already implemented as a test equation in TerraDG. We add the elastic wave equation following the general TerraDG structure given by [3]. For every equation the method `evaluate_flux()` has to be provided, which for the elastic wave equation is defined according to the equation system (2.7). The specified traction at the boundary is set to be zero and is implemented following the definition given in (2.9).

4.2.1. Differentiate-first-then-discretize

For the *differentiate-first-then-discretize* ansatz, the adjoint equation is implemented the same way as the forward system. For the advection equation we derived the analytical adjoint equation in (3.19a) as

$$
\begin{aligned}
-\frac{\partial \Lambda}{\partial t} - \beta \nabla \Lambda = \gamma_1(\phi - \Phi) \quad (T, 0) \times \Omega \\
\Lambda = 0 \quad (T, 0) \times \Gamma \\
\Lambda = \gamma_2(\phi \mid_{t=T} - \Phi \mid_{t=T}) \quad \Omega \text{ at } t = T.
\end{aligned}
$$

and analytical gradient formulation as

$$\frac{DJ}{D\beta} = \int_\Omega \int_0^T \Lambda \cdot \phi_\beta \; dt d\Omega.$$

This is the gradient when computing the cost using the time tracking cost functional J_T as is given in (3.1). If the cost is computed only at the final time using J_C as defined in (6.1), the source term of the adjoint equation becomes zero and the total derivative becomes

$$\frac{DJ}{D\beta} = \int_\Omega \Lambda \mid_{t=T} \cdot \phi_\beta \mid_{t=T} \; dt d\Omega.$$

During the forward solve we store the difference $(\phi - \Phi)$ in every time step and can then use it to compute the source term of the adjoint equation. In the reverse solve the recorded values are used to compute the adjoint equation. To solve the advection equation with a source term, we use the Fractional-Step Method as shown in [8]. First the homogeneous PDE

$$\Lambda_t^* + \nabla \cdot F(\Lambda^*) = 0 \tag{4.3}$$

then the ODE equation

$$\Lambda_t = \gamma_1(\Lambda^* - \Phi) \tag{4.4}$$

with the source term as the right hand side. In TerraDG we solve the ODE using Euler time stepping, following the example given in [8]. For the derivation of the analytical adjoint equation we assumed that the boundary g is independent of the parameter and its derivative with respect to the parameter is zero. As we apply the exact solution at the boundary for the test example, this condition is slightly different. For the reverse adjoint computation we assume that the boundary stays the same and apply the periodic boundary condition to the homogeneous adjoint PDE system.

Once the adjoint equation is computed, the gradient can be computed using the derived analytical integral formulations. However, the analytical formulation includes the solution of the sensitivity equation ϕ_β. We compute this during the forward solve using a simple Finite Differences approximation. This means that the forward model solves the advection equation twice: $\phi(\beta)$ and $\phi(\beta + \epsilon)$. The reverse adjoint solve also solves one PDE advection equation. In total, we require three PDE solves to compute the gradient. In theory, we should also be able to use the right side of the Equation 3.21a to replace the sensitivity derivative.

For large systems or long time simulations storing the forward solutions at every time step would of course quickly become infeasible memory wise. A better approach is to use a check pointing algorithm such as [29]. The main idea of these are to stores the solution only at a few optimally placed check points and can use these to compute the reverse solve more efficiently. As check pointing is not implemented for the adjoint solver, we restrict our self to short simulation times and smaller test sizes.

4.2.2. Discretize-first-then-differentiate using AD

The general approach of AD is to decompose the code of a function into the most elementary operations and functions for which the analytic derivative can be computed [30]. AD then computes the derivative of a function by applying the chain rule using ether *forward-mode* or *reverse-mode*. Before explaining how AD is applied to the DG code TerraDG, we look at a simple *forward* AD example to understand the general approach.

Forward mode automatic differentiation (AD)

The simple function $f = sin(2\pi(x_1 - \beta t))$ which can be considered as a composite function of the most basic operations such as sin, cos or $+/*$ [30]. We follow the same steps as for the example in [30]. Equation (4.5) shows the basic operations of f in the left column and the derivative of each operation on the right.

$$\begin{aligned}
z_1 &= x & z_1' &= x' \\
z_2 &= \beta & z_2' &= \beta' \\
z_3 &= t \cdot z_2 & z_3' &= t \cdot z_2' \\
z_4 &= (z_1 - z_3) & z_4' &= (z_1' - z_3') \\
z_5 &= sin(2\pi z_4) & z_5' &= cos(2\pi z_4) \cdot 2\pi \cdot z_4'
\end{aligned} \tag{4.5}$$

Starting from the most outside composite derivative, the complete derivative is put together in one forward sweep per parameter. If we vary the function f with respect to x, then in equation (4.5) we would know that the derivatives are $z_1' = x' = 1$ and $z_2' = \beta' = 0$. This can be seen as setting a seed $(1, 0)$, where only the derivative variable is one and all others are zero [30]. If the derivative f_β is to be computed, then the forward sweep is made with $(x' = 0, \beta' = 1)$.

$$\begin{aligned}
f_x &= z_{5_x} & f_\beta &= z_{5_\beta} \\
&= cos(2\pi \cdot z_4) \cdot 2\pi \cdot z'_{4_x} & &= cos(2\pi \cdot z_4) \cdot 2\pi \cdot z'_{4_\beta} \\
&= cos(2\pi(z_1 - z_3)) \cdot 2\pi \cdot (z'_{1_x} - z'_{3_x}) & &= cos(2\pi(z_1 - z_3)) \cdot 2\pi \cdot (z'_{1_\beta} - z'_{3_\beta}) \\
&= cos(2\pi(z_1 - t \cdot z_2)) \cdot 2\pi \cdot (1 - t \cdot 0) & &= cos(2\pi(z_1 - t \cdot z_2)) \cdot 2\pi \cdot (0 - t \cdot 1) \\
&= cos(2\pi(x - t \cdot \beta)) \cdot 2\pi & &= cos(2\pi(x - t \cdot \beta)) \cdot 2\pi \cdot -t
\end{aligned} \tag{4.6}$$

The Julia *forward-mode* AD package `ForwardDiff` uses the dual numbers approach, which transforms all numbers into a dual number containing the actual function value a and the derivative component b of the function as a second additional component [31]. Mathematically, the dual number approach for computing the derivative is defined as

$$f(a + \sum_{i=1}^{N} b_i \epsilon_i) = f(a) + f'(a) \sum_{i=1}^{N} b_i \epsilon_i, \tag{4.7}$$

where f is a elementary function for which the overloaded derivative is defined [31]. For the type overloading to work, all data structures in the TerraDG code case which are used in the gradient computation have to be changed to use the abstract super type `Core.Real` as this is also the super type of the ForwardDiff `Dual{T,V<:Real,N}<:Real`. As shown in [31], the `Julia ForwardDiff` package provides the three methods:

- `derivative(f, x)` for $f(x) : \mathbb{R} \to \mathbb{R}$
- `gradient(f,x)` for $f(x_1, ..., x_n) : \mathbb{R}^n \to \mathbb{R}$
- `jacobian(f,x)` for $f(x_1, ..., x_n) : \mathbb{R}^n \to \mathbb{R}^n$

Gradient computation using AD

The ansatz *discretize-first-then-differentiate*, means that the AD routines are applied to the discrete implementations of the cost function and the DG implementation of the PDE. TerraDG already provides the routines for the PDE discretization and the forward time loop. The cost function is easily computed using numerical quadrature as

$$J(\phi_i, t^k, \beta) = \frac{\gamma_1}{2} \Delta t \sum_{k=0}^{m} \Delta x \sum_{i=1}^{n} (\phi_i^k(\beta) - \Phi_i^k(\hat{\beta}))^2 \tag{4.8}$$

$$+ \frac{\gamma_2}{2} \Delta x \sum_{i=1}^{n} (\phi_i^m(\beta) - \Phi_i^m(\hat{\beta}))^2. \tag{4.9}$$

The computation of the total derivative can be computed following [15] using AD is divided into the two main steps:

1. **Forward solve:**
   a) Compute the forward solution ϕ of the PDE
   b) Compute the partial derivatives $\frac{\partial J}{\partial \phi}$, $\frac{\partial F}{\partial \phi}$ and $\frac{\partial J}{\partial \alpha}$,$\frac{\partial F}{\partial \alpha}$ using an AD routine
2. **Backward solve:**
   a) Solve the adjoint equation (3.23) for Λ
   b) Compute the total derivative using (3.26)

Computing the partials using AD

In order to compute the partial derivatives listed in step 1.b), the code functions must be passed in the right shape to the `ForwardDiff` package. In the TerraDG code base, the unknown state variable ϕ is stored in the three dimensional array `dofs`. The first dimension contains the basis function points, e.g. for order two this would result in $ord = 4$ basis points. The second dimension gives the number of variables $nvars$ of the PDE equation. The third dimension now contains the cell values of the row-wise flattend two dimensional discrete grid of length $ndofs$. Following the iterative scheme derived in (3.58) using [15] the two derivatives $\frac{\partial F^k}{\partial \phi^k}$ at the dofs time level k and $\frac{\partial F^k}{\partial \phi^{k-1}}$ at the dofs time level $(k-1)$ have to be computed. For Euler time stepping, the method `step` computes `dofs .+= dt * dofsupdate`, where the `dofsupdate` is the update $\delta\phi$ computed by the TerraDG method `evaluate_rhs`. As shown in (3.23) the derivative at the new time level k is independent of the discretization, being the diagonal matrix with $1/dt$ as a value. The derivative at time level $(k-1)$ depends on the discretization and needs to be computed using AD. The output of the dofs update method is the same size as the dofs array $D = (ord \times nvars \times ndofs)$, meaning the forward differentiation method `jacobian` is required for the derivative. If the `dofs` array of size D is directly passed to the gradient as the argument `x`, then the computed derivative matrix would be a very large Jacobian matrix of size $(D \times D)$. Many of the computed partials inside the Jacobian are however not wanted. For example, the a PDE with variables ϕ_1, ϕ_2 and ϕ_3, the full Jacobian would contain the cross derivatives between the variables e.g. $\frac{\partial F_{\phi_1}}{\partial \phi_2}$. To compute only partials of `evaluate_rhs` for a given variable and basis function point with respect to only the corresponding section in the dofs vector, a wrapper method is required. As shown in the listing (4.1), the methods simply reassembles the dofs matrix. However, the wrapper function can now be passed as the function handle to `ForwardDiff.jacobian` with an explicit order and variable index, which means only that subsection of the dofs vector is used as x. The computed derivative matrix is also sparse, as was shown in (4.8). The Jacobian is stored using `SparseVectors` and the computation is done only once per forward solve, as the derivative matrix does not change over the time levels. Using a similar wrapper function setup, the derivatives of J with respect to the dofs vector can be computed using the `gradient` AD routine.

With the functions for computing the partials defined, the main time loop is extended by two structs: one `CostFunction` containing the methods for computing the cost function and `AdjointPartials` for computing the derivatives as explained above. The backward solve of the adjoint variable is computed using the iterative scheme (3.58). The implementation is straight forward: one time loop and inside two for loops over the order and the variables. After the forward and backward solve is done, the total derivative can be computed using the formula derived in (3.23), as the dot product of the vectors Λ and the derivative $\frac{\partial F}{\partial \alpha}$ plus the derivative of the cost function w.r.t. the parameter α.

```
function evaluate_rhs_diff_dofs(dofs, ..., var_idx, ord_idx, diff_dof)

    du, dofs_diff = similar(dofs), similar(dofs, size(dofs))
    du .= 0.0
    for ord in 1:size(dofs)[1]
        for var in 1:size(dofs)[2]
            if ord == ord_idx && var == var_idx
                dofs_diff[ord_idx, var_idx, :] .= diff_dof
            else
                dofs_diff[ord, var, :] .= dofs[ord, var,:]
            end
        end
    end

    f = (dofs_diff_, du_) -> evaluate_rhs(... dofs_diff_, ...)
    return (-1/dt).*dofs_diff[ord_idx,var_idx,:]
            -f(dofs_diff, du)[ord_idx,var_idx,:]
end

wrhs = ddof -> evaluate_rhs_diff_dofs(...)
drhs_ddofs = (dofs,.., var, ord, ddof) -> jacobian(w, ddof)
```

Source Code 4.8.: Wrapper functions for the AD forward diff routines to avoid unnecessary cross derivatives.

```
for ord in 1:order
    for var in 1:nvars
        if is_final_time
            tmp = sparse(self.drhsddofs(dofs, ...,
                        var, ord, dofs[ord,var,:]))
            tmp[diagind(tmp)] .+= 1.0/dt
            tmp .*= (-1.0)
            self.A1_trans[ord, var] = transpose(tmp)
        end

        # derivative of J w.r.t. dofs
        dJddofs_k[ord, var, :] = self.dJddofs(...,
                                     dofs[ord,var,:])

        # derivative of rhs w.r.t. parameter
        drhsdparams_k[ord, var, :] = t.*self.drhsdparams(...,
                                     parameters[var])
    end
end
```

Part IV.

Results

5. Tests

5.1. Convergence tests

Both the forward solution model and gradient can be checked with convergence tests. For the forward model we know the time-stepping order which must be achieved. As the LOH1 equation does not have a known analytic solution, we test the planar wave set up to verify that the solutions are computed correctly. To verify the gradient computed, we use the Taylor test routine provided by dolfin-adjoint [7]. To check the gradient for the advection equation example computed using the TerraDG implementation, we can compare to the exact solution.

5.1.1. Test equation convergence rates

For the verification of the forward models, we compute the convergence rates and orders using the test cases where an analytical solution is given. For a Runge-Kutta time-stepping scheme with s stages, the convergence rate and the order are computed using

$$\text{conv. rate} = \frac{e_1}{e_2} \qquad \text{conv. order} = \frac{\log(\frac{e_1}{e_2})}{\log(\frac{\Delta t_1}{\Delta t_2})} \tag{5.1}$$

where e_1 is the L_2-error computed with step size Δt_1 and e_2 is computed with half the step size $\Delta t_2 = \frac{\Delta t_1}{2}$. For the advection equation (2.6) the analytic solution was given as:

$$\phi(t, \mathbf{x}) = sin(2\pi(x - \beta_x t)). \tag{5.2}$$

We remind our self, that for the two dimensional planar wave problem we derived the analytic solution as:

$$\phi(t, \mathbf{x}) = \begin{bmatrix} -k\sin(2\pi(x - c_p t))(\lambda + 2\mu) \\ -k\sin(k(x - c_p t))\lambda \\ -k\sin(k(x - c_s t))\mu \\ c_p k\sin(k(x - c_p t)) \\ c_s k\sin(k(x - c_s t)) \end{bmatrix}, \tag{5.3}$$

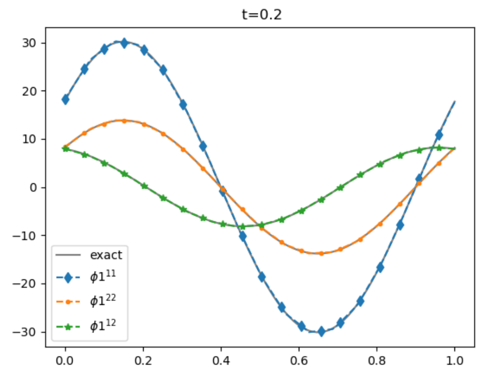
where $k = 2\pi$ and c_p, c_s are the wave speeds. Table (5.1) shows the achieved convergence rates and orders for the advection equation. The mesh points in every iteration is doubled, in order to half the cell size Δx / Δy. For the advection equation we can therefore show that the forward solver computes the solution correctly.

DF	L_2-error	conv. rate	conv. order	L_2-error	conv. rate	conv. order
96	0.00823414	1.99001704	0.99278079	0.00196984	3.98674329	1.99521071
192	0.00412927	1.9940917	0.99573176	0.00049292	3.99631422	1.99867002
384	0.00206826	1.99649125	0.99746675	0.00012326	3.99911332	1.99968016
768	0.00103524	1.99785882	0.99845464	3.082e-05	3.99971794	1.99989827
1536	0.00051797	1.99865656	0.99903059	7.7e-06	3.99994363	1.99997967

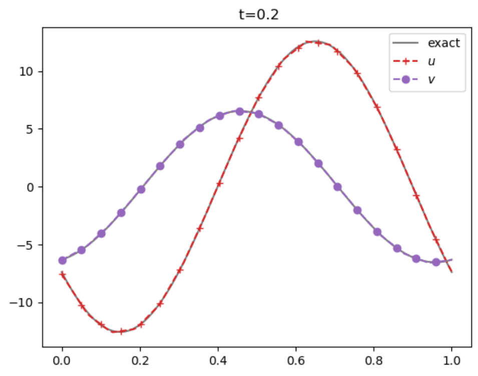
(a) Runge-Kutta order 1 (b) Runge-Kutta order 2

Table 5.1.: Convergence rates and orders of the Runge-Kutta time discretization scheme for the advection equation.

For the elastic wave equation we at first faced some stability problems. In tests, we can also see that the computation requires a very small time step. The CFL number suggested in [12] is 0.01 if the domain is divided into 7 or 8 number of cells. If we use the by [12] recommended magnitude of the CFL condition, 0.01, we see that the numerical solution converge towards the exact solution. The second order Runke-Kutta convergence rate is ≈ 3.63978 and with that clearly better than the first order scheme but not the exact convergence factor 4. The Figure 5.1 show the DG approximations of the different components compared to the exact solutions. However, the large number of time steps make the computation time for the non-optimized FEniCS model quite long.



(a) The stress component solutions when using a time step of $\Delta t = 0.0003125$.



(b) The velocity component solutions when using a time step of $\Delta t = 0.0003125$.

Figure 5.1.: DG approximation and exact solution for the planar elastic wave equation in 2D. The plot shows a one dimensional slice at y=1/2. The CFL condition is chosen small enough, such that the approximation converges towards the exact solution.

equation	$\mathcal{M}$	$\mathcal{D}$	$\mathcal{K}$	dofs	L_2-error	J
elastic planar 2D	8	1	1920	2	0.805048268826636	664.658074451846
elastic planar 2D	16	1	7680	2	0.22118072435718	678.229438053766

Table 5.2.: Approximation L_2-error for the elastic planar wave equation.

5.1.2. Gradient convergence rates

For testing the gradient computation, dolfin-adjoint provides a Taylor Test routine, as the total derivative $\frac{DJ}{D\alpha}$ can also be approximated using the second order Taylor remainder [2]. This test routine be used to can simply verify that the adjoint computation of dolfin-adjoint is working correctly. If the Taylor test fails, it may be that the dependencies of the control variable are not properly linked. For a second order approximation

$$T_{\mathcal{O}(2)} = |\hat{J}(\alpha + h\delta) - \hat{J} - h\nabla\hat{J}\delta|, \tag{5.4}$$

the residual should decrees by a factor of four each time the perturbations δ is halved. For the FEniCS LOH1 and advection models the Taylor test routine is used to check if the gradient computation achieves the correct second order convergence. The step size h should ideally be chosen to be of the same magnitude as the control variable [7]. Table (5.3) shows a selection of different optimization settings and the achieved convergence orders. For the elastic wave test example, the planar wave setting, the Taylor test was not made, as the computation time for that model is quite large. The problem for this example case is, that an increasing number of time steps significantly increases the computation time when computing the gradient with dolfin-adjoint.

Equation	DF	$\mathcal{D}$	$\mathcal{K}$	control	mean convergence rate
advection-2D	384	1	rk1	β_x	1.99703241430964
advection-2D	768	2	rk1	β_x	1.99753701617741
Advection 2D	384	1	rk2	β_x	1.99787865647545
Advection 2D	768	2	rk2	β_x	1.99792167791555
LOH1	7680	1	rk1	s_x	1.99996326615342
LOH1	15360	2	rk1	s_x	1.99996620106436

Table 5.3.: The FEniCS Taylor test routine convergence orders can be used to validate the gradient computed for the test equations. The convergence order must be two, other wise the gradient computation still contains errors. The type of cost function used was the controllability functional J_C.

5.1.3. Convergence test for the TerraDG adjoint solver

In general, the cost functional depends on the parameters α directly and indirectly via the state vector. The reduced cost functional $\hat{J}$, re-formulates the cost functional so that the indirect dependency on parameters α become a direct dependencies [2]. To get comparison gradient approximation, we can implement the simple FD scheme

$$\frac{\partial \hat{J}}{\partial \beta} \approx \frac{\hat{J}(\beta+\epsilon) - \hat{J}(\beta-\epsilon)}{2\epsilon}$$

for computing the gradient using the analytic solution of ϕ [2]. Table (5.4) shows the approximated gradients.

N	J type	J	DJ
100	J_C	0.2048959051063637	2.5161692718895927
200	J_C	0.20550163948006359	2.5288771975051914
400	J_C	0.2058045066669135	2.535231160312998
80	J_T	0.28404712098242624	3.513347722958295
120	J_T	0.2822130575671631	3.4964016316303597
200	J_T	0.2807607164159079	3.482960072540482

Table 5.4.: Finite Difference approximation of the Reduced functional gradient $\partial\hat{J}/\partial\beta$.

We can simply check if the Finite Difference method computes the correct gradient value. If we consider the controllability cost function

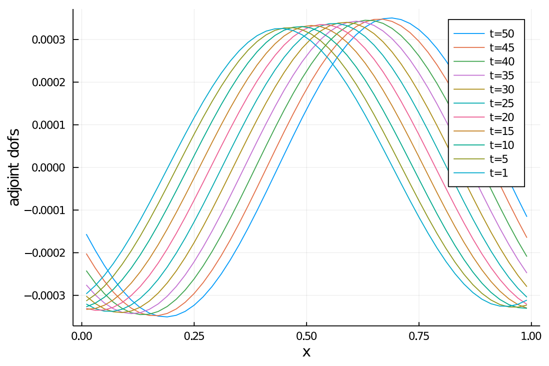
$$J_C = \frac{1}{2}\int_{\Omega} (\phi(\beta) - \varPhi)^2 dx, \tag{5.5}$$

we can insert the analytic solution and can thus write the cost functional in reduced form. The gradient of $\hat{J}_C$ can be derived analytically using the chain rule and then computed using numerical quadrature as

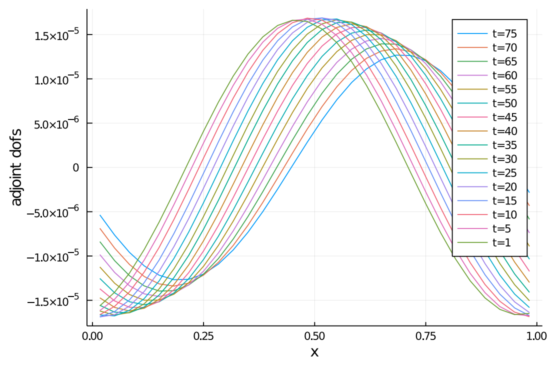
$$\begin{aligned} \frac{d\hat{J}_C}{d\beta} &= \int_0^1 [\sin(2\pi(x-\beta t_n)) - \sin(2\pi(x-\hat{\beta} t_n))] \\ &\quad \cdot (-2\pi t_n \cos(2\pi(x-\beta t_n))) \quad dx \\ &\approx 2.541636269831977 \end{aligned} \tag{5.6}$$

Comparison to the gradient values computed using AD

In Table (5.5) we see the gradient values computed using the AD adjoint implementation in TerraDG. For the controllability cost functional the computed values seem to converge towards the FD solution. However, as the forward model suffers from strong numerical diffusion, the gradient value is of course affected by this. We can also see the loss in accuracy in the cost function value J_C. To overcome the problem of numerical diffusion, one could simply increase the DG order. However, when computing the adjoint using higher



(a) Adjoint for J_C



(b) Adjoint for J_T

Figure 5.2.: Solutions to the adjoint equation computed using automatic-differentiation. The Controllability cost functional J only computes the cost at the final time step, making the adjoint an homogeneous advection equation. The Timetracking cost functional computes the cost over the entire time domain, resulting in a source term for the adjoint equation.

M	DG	RK	L_2-error forward sol.	J type	J	DJ
30	1	RK1	0.105087624845139	J_C	0.179632166602784	2.11090956563557
60	1	RK1	0.054493340271583	J_C	0.191903487169665	2.31334264931309
30	1	RK1	0.105087624845139	J_T	0.244841372899244	3.47921711786876
60	1	RK1	0.054493340271583	J_T	0.259702163311143	3.79371286885047

Table 5.5.: Gradients computed using TerraDG using the AD ansatz.

order DG elements, the adjoint solution becomes zero as can be seen in Figure (5.3). The adjoint computation should stay exact the same for higher order. No literature example could be found, regarding an implementation of an adjoint solver using automatic differentiation and higher order DG. A possible cause for the problem could be that the higher order volume term is not correctly differentiated by the AD package. If more time for the project been available, the results could have been compared with a different AD packages e.g. reverse mode AD. The gradient for the Time tracking cost functional J_T computed using AD in TerraDG also contains a large error when comparing to the Finite Differences approximation. This could be due to the stronger effects of numerical diffusion when computing the cost over the complete time domain or could of course be an indicator that the adjoint is not jet computed correctly. Further test would have to be made here. Especially, a test using higher order DG elements would be good, to get a better forward solution approximation.

The AD approach currently uses the sequential forward solver. As there are still issues with the gradient computation, code optimizations where not jet made. We can still look at where in the adjoint solver the most compute time goes to. For the AD approach the most expensive operation is computing the adjoint system matrix $\frac{\partial F}{\partial \phi}$, as it has $(n^2 \times n^2)$. The computation time for the computed gradients are listed in Table (5.6). The run time of the AD adjoint solver in TerraDG is dominated by the computation of the gradient matrix $\frac{\partial F}{\partial \phi}$. The time and memory measurement of the AD computation shows how costly computing the full Jacobian matrix is. For twice as many degrees of freedom, the computation time is approx. 10 times as long. For this reason, using higher order DG might be advantageous. As the cause for the higher order solution converging towards zero could not be found jet, this theory could not be checked.

Despite the large error in the gradient, it is still possible to use the computed gradient in a Optimal Control Optimization. We use the simple the advection equation test case: $\phi(x,t) = sin(2\pi(x - \beta t))$. The mock measurements are created with the velocity $\beta = 0.1$. Starting with the initial guess $\beta^0 = 0.25$, the optimization convergences in five iterations towards $\approx 0.09976115231365439$.

solver step	M	time and storage measurement
Controllability J_C	30	
dFdphi		12.294389 s (103.00 M alloc: 6.435 GiB, 10.01% gc t)
forward		58.433055 s (304.08 M alloc: 12.601 GiB, 5.26% gc t)
reverse		5.533887 s (11.10 M alloc: 844.495 MiB, 5.06% gc t)
Controllability J_C	60	
dFdphi		121.994853 s (1.57 G alloc: 98.896 GiB, 14.03% gc time)
forward		260.195863 s (2.78 G alloc: 129.595 GiB, 9.10% gc time)
reverse		33.423291 s (79.03 M alloc: 16.776 GiB, 4.13% gc time)
Time Tracking J_T	60	
dFdphi		132.404556 s (1.56 G alloc: 98.622 GiB, 11.62% gc time)
forward		366.741602 s (4.20 G alloc: 213.477 GiB, 11.23% gc time)
reverse		8.740723 s (81.41 M alloc: 2.395 GiB, 8.20% gc time)

Table 5.6.: The run time of the AD adjoint solver in TerraDG is dominated by the computation of the gradient matrix $\frac{\partial F}{\partial \phi}$. The time and memory measurement of the AD computation shows how costly computing the full jacobian matrix is. For double the number of cells, the computation time is approx. 14 times as long.

Comparison to the gradient values computed using the Differentiate-then-discretize Ansatz

The second adjoint method for computing the gradient is the *differentiate-then-discretize* approach. The computed adjoint equation when using the Time tracking cost functional J_T can be seen in Figure (5.4). The adjoint equation solution is nearly the same as for the AD computation, however the solutions differ in the magnitude. This is because both

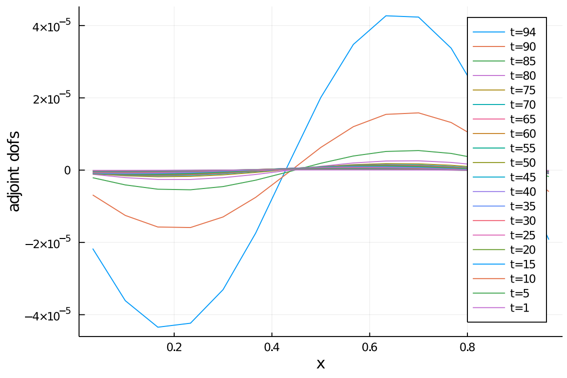


Figure 5.3.: The adjoint equation computed using ForwardDiff AD becomes zero, when switching to higher order DG element. The exact cause for this error could not be determined. The correct adjoint should also be a sinus curve which is advected to the left.

methods use a differently scaled terminal condition on the adjoint. The AD Ansatz uses $\Lambda^T = \Delta t \frac{\partial J}{\partial \phi}$, while in the differentiate then discretize Ansatz we solve the adjoint equation using the TerraDG solver. We remind ourself, that the adjoint of the advection equation looks like

$$-\frac{\partial \Lambda}{\partial t} - \beta \nabla \Lambda = \gamma_1(\phi - \Phi) \quad (T, 0) \times \Omega$$

and uses the terminal condition $\Lambda = \gamma_2(\phi \mid_{t=T} - \Phi \mid_{t=T})$. The gradient values computed using the second ansatz in TerraDG, are listed in Table (5.7). We see the same again: the error in the gradient is quite high, mostly due to numerical diffusion.

M	error forward	J	DJ
30	0.10508762484513891	0.244841372899244	3.027224915693371
60	0.05449334027158299	0.259702163311143	3.1923370015307433

Table 5.7.: Gradients computed using the *differentiate-then-discretize* using TerraDG for the Timetracking cost functional. The error of the forward

Just like for the AD Ansatz, the implementation for the differentiate-then-discretize is not yet optimized for run time or memory requirements. In Table 5.8 we can see the memory and run time for the sequential forward and reverse models. The forward solve time increases much stronger than the reverse solve with increasing degrees of freedom. This could be due to the storage of the forward solutions. As mentioned before, a check pointing algorithm such as [29] could improve the performance of the adjoint solver.

solver step	M	Time and memory measurements
forward	15	30.708099 seconds (63.73 M alloc:: 2.677 GiB, 3.80% gc time)
reverse	15	4.652970 seconds (10.92 M alloc: 355.035 MiB, 2.95% gc time)
forward	30	46.934199 seconds (167.45 M alloc:: 5.099 GiB, 3.87% gc time)
reverse	30	9.411440 seconds (59.81 M alloc:: 1.473 GiB, 4.21% gc time)
forward	60	177.796064 seconds (986.43 M alloc:: 24.047 GiB, 5.02% gc time)
reverse	60	54.219801 seconds (448.45 M alloc:: 10.386 GiB, 5.10% gc time)

Table 5.8.: Time and memory measurements for the gradient computation using the *differentiate-then-discretize* Ansatz in TerraDG and a Time Tracking cost functional J_T. The allocations are given in M

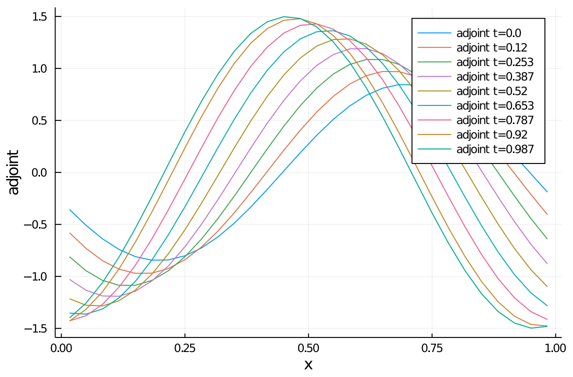


Figure 5.4.: The adjoint computed using the *differentiate-then-discretize* approach which solves the discrete analytical adjoint equation using the same solver as the forward problem. For the solver, the time stepping is not reverse but the advection velocity is reversed instead with the time direction remaining forward in time stepping.

5.2. Visualization

5.2.1. Plots of the approximated solutions

It is always a good correctness check to plot the PDE model solution. For the advection equation we can run the simple Parameter Optimization Problem as explained previously. We plot the solutions in every iteration, as shown in Figure 5.5. For the LOH1 we can see the computed solution in Figure (5.6) and (5.7).

5.2.2. Graph visualization methods in FEniCS

To visualize how dolfin-adjoint internally processes a given model internally, it is possible to create a graph showing the basic operations and dependencies using Graphviz (or Ten-

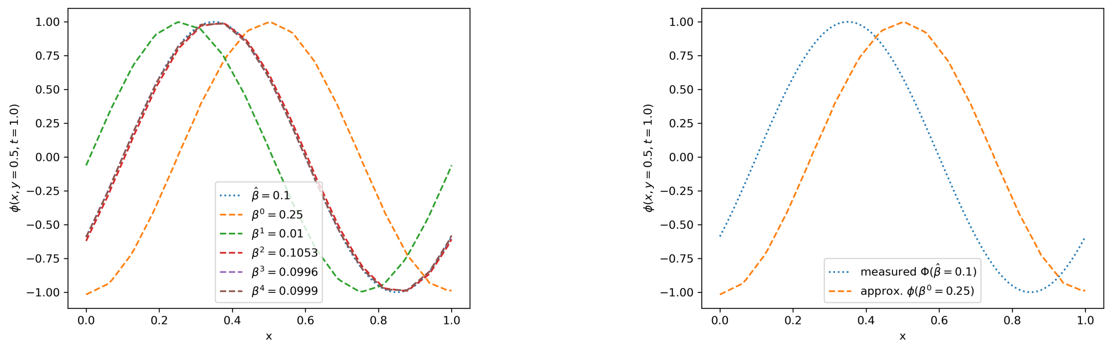


Figure 5.5.: Correctness check: Plotting the solution of the advection equation, computed using the FEniCS model. The plots are a line plot over x at y=0.5. At each iteration, the new forward solution is plotted. As we can see, the optimization converges quite quickly towards the correct measurement data.

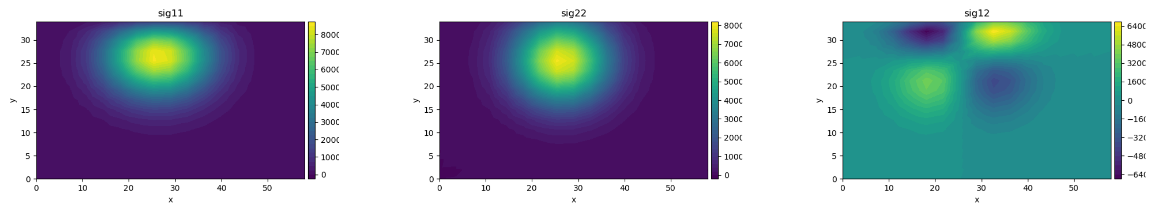


Figure 5.6.: Stress component solution of the LOH1 at the final time $t = 1.0$. The initial disturbance is set as a Gaussian curve as shown in equation 2.15. The layer region is 1[km] deep from the top boundary. The source origin is at (26km, 32km) We can see that the zero boundary condition in ϕ^{12} is still approximately full filled.

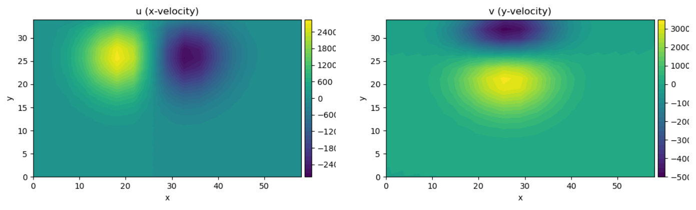


Figure 5.7.: The velocity components of the LOH1 equation at the final time t=0. The layer region is 1[km] deep from the top boundary. The source origin is at (26km, 32km).

sorFlow) [7]. By adding the two visualization method calls

```
    tape = get_working_tape()
    tape.visualize('graph.dot')
```

all connections are recorded in a dot file which is visualized in Figure (5.8). The graph shows the simulation flow for two time levels and the correct dependency of the flow velocity `beta_vec` on the x and y component which can be used as control variables.

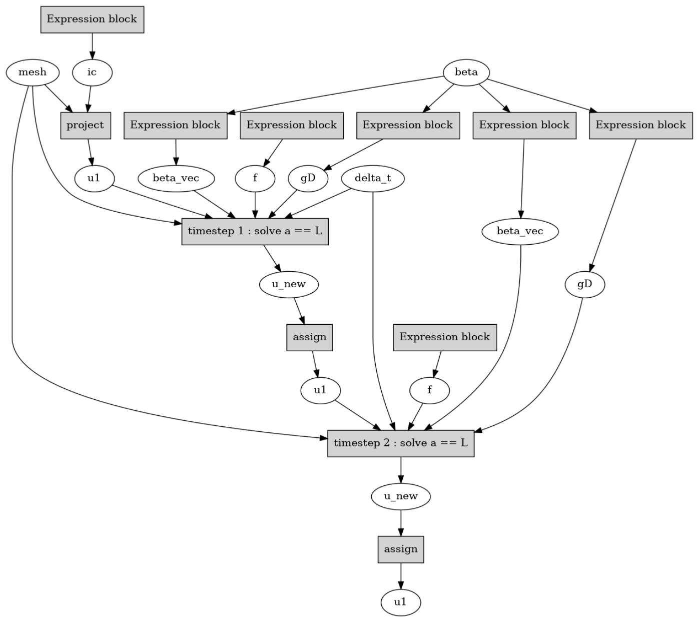


Figure 5.8.: Visualization of the dependencies graph generated by dolfin-adjoint showing two time steps of a forward solve.

6. Numerical Results

6.1. Paremeter optimization results

In this section we now want to use the adjoint solver to solve optimal control problems. The main model settings are: DG-order $\mathcal{D} = [1, 2]$, mesh size $\mathcal{M} = [8, 16]$ and Runge-Kutta order $\mathcal{K} = [1, 2]$. The degrees of freedom of the PDE solver can be computed as $df = \mathcal{D} \cdot \mathcal{M}^2$. For the example optimizations the Quasi-Newton minimization routine L-BFGS-B is used.

6.1.1. Optimization example 1: Initial disturbance origin of LOH1

The LOH1 setup simulates a basic earthquake scenario, where the initial disturbance is located in the half-space with the origin $s_m = (sx, sy)$. Mock measurements are created using the origin $s_m = (26, 32)$ to act as placeholder for real sensor data. We can use the exact origin point of the measured data to check the correctness of the optimization. For the first optimization example, the controllability cost function J_C

$$J_C(\phi, \alpha) = \frac{\gamma_2}{2} \int_\Omega (\phi \mid_{t=T} - \Phi \mid_{t=T})^2 d\Omega. \tag{6.1}$$

as defined in (6.1) is used to determine the earthquake origin (s_x, s_y) of the LOH1 model. We further restrict the optimization by passing the domain boundaries as optimization bounds. The mock measurements ϕ_m are computed using the same DG-order and mesh points as the simulated solution. To compare with a more realistic cost function setting, we use the sensor cost function J_S, where the measurements are taken at a limited number of receiver points. For the computation in FEniCS we define the discrete sensor cost function as:

$$J_S(\phi, \alpha) = \frac{1}{2} \sum_{k=0}^{M} \int_{\Gamma_{top}} (\phi(x_{i,j}, t) - \Phi(x_i, t))^2 d\Gamma_{top} dt \tag{6.2}$$

$$+ \int_{\Gamma_{top}} (\phi(x_{i,j}, T) - \Phi(x_i, T)^2 d\Gamma_{top}. \tag{6.3}$$

For this example, we compute the cost function J_S over the points at the top boundary $\Gamma_{top} = [x, y = 33]$. To see the stabilization effect of the second additional integral term at the final time T, we also do the optimization with $\gamma_2 = 0$ and name the resulting cost function for simplicity J_K. Table (6.1) and Figures show the optimization results for the solver parameters DG order $\mathcal{D}$, number of cells both directions $\mathcal{M}$ and the Runge-Kutta

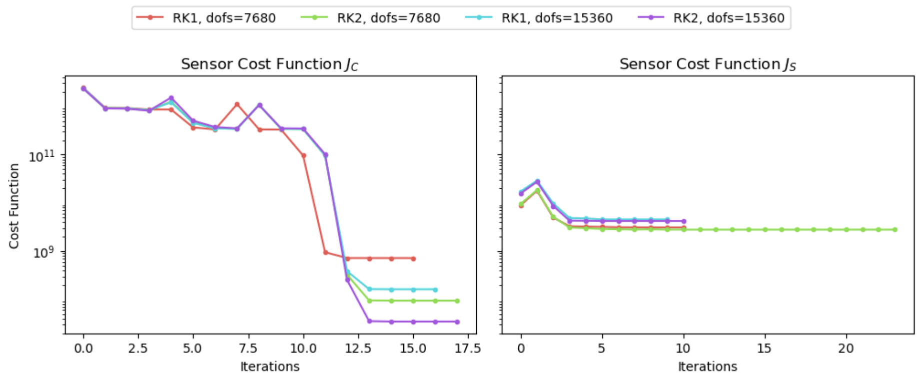


Figure 6.1.: Comparison of the controllability cost functional value J_C where the cost functional is computed over the entire domain and the sensor cost functional value J_S, where the cost is computed only at the upper boundary but over the entire time domain. .

order $\mathcal{K}$. As expected, the controllability cost functional J_C performs best. For the sensor cost functional J_S to achieve a similar error, second order DG-element and second order time stepping is required. In Figure (6.2) we see the cost over the optimization iterations for the controllability cost functional and the sensor cost functional. As expected, J_C requires more iterations to reduce to zero. The sensor cost functional value drops only very slightly at the beginning of the iteration but then stays constant.

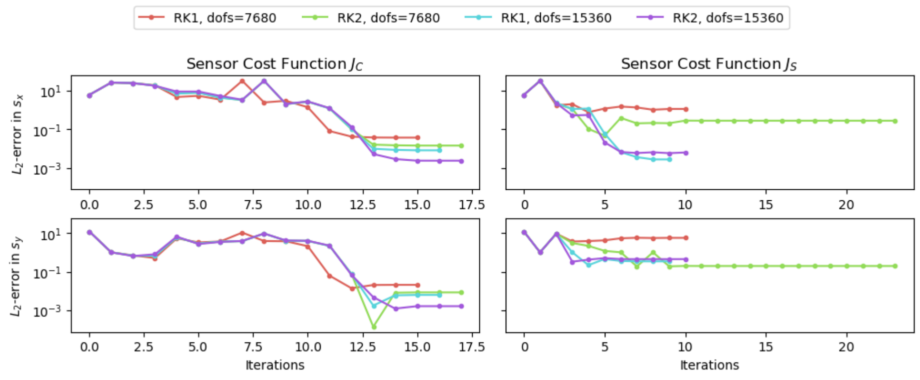


Figure 6.2.: Error in the optimization parameters (s_x, s_y) in the x and y direction. The cost functional J_C computes the cost over the entire computation domain Ω. The errors on the right are for the sensor cost function J_S. We see that the optimization still performs quite well, even if we compute the cost only at the top boundary over the entire time domain.

The error graphs, as shown in (6.2), show how well the optimization routine estimates

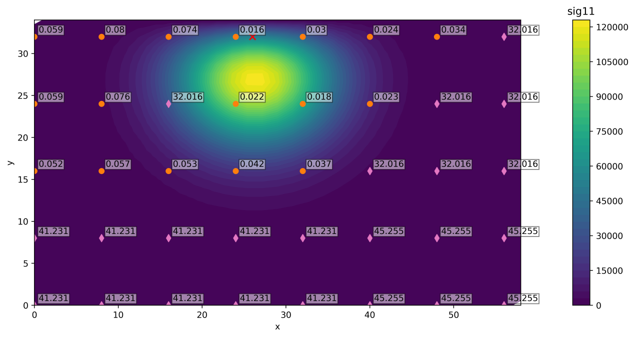


Figure 6.3.: Variation of the initial guess for the Optimal Control Problem. One can see that the optimization may not converge, if the initial error is too large. The variations where computed for a course grid of $\mathcal{M} = 8$, $\mathcal{K} = RK1$ and $\mathcal{D} = 1$.

the optimal parameter. For the cost functional the error decreases more slowly but then drops lower than the sensor cost functional. For J_S to achieve the same error as in J_C, higher order DG element and higher order time stepping has to be chosen. The gradient development over the iterations is shown in (6.4).

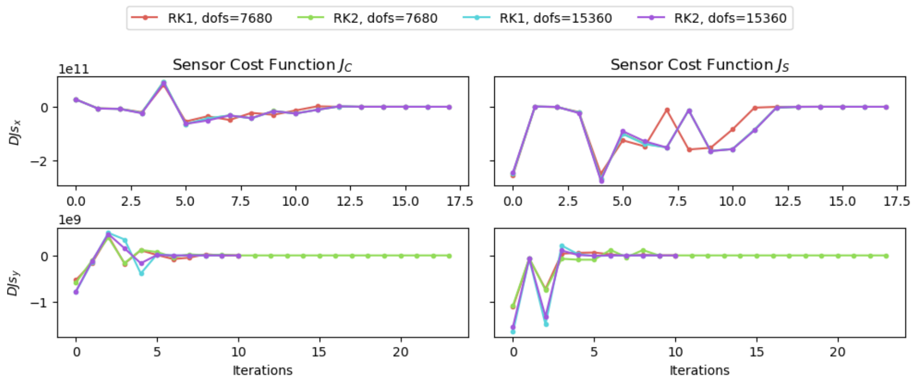


Figure 6.4.: Gradient of the cost functional for the optimization parameters (s_x, s_y).

The initial value could of course have a large impact on the convergence towards the correct origin point. As a test, we start a Optimal Control Optimization with different initial values. The initial values are depicted in Figure (6.3), where each initial start point is coloured, orange for convergence and blue for divergence. The final error is noted next to the iteration starting points. We can also compare to the cost functional J_K, where we did not add the integral at the final time again. According to [1], it can acct as a stabilization term. In Table (6.1), we can see that the is always higher apart from when using the coarsest model settings (7680 dofs with RK1).

$\mathcal{M}$	$\mathcal{D}$	$dofs$	$\mathcal{K}$	type J	Tit	final error	opt param
16	1	7680	RK1	J_C	16	0.04188	(25.96357 32.02065)
16	1	7680	RK1	J_K	10	1.0779	(25.08867 32.57563)
16	1	7680	RK1	J_S	11	5.74064	(24.88569 26.36854)
16	**1**	**7680**	**RK2**	J_C	**18**	**0.01639**	**(25.98592 32.00838)**
16	1	7680	RK2	J_K	36	0.72785	(25.79932 31.30036)
16	1	7680	RK2	J_S	24	0.33689	(25.72567 32.19555)
16	**2**	**15360**	**RK1**	J_C	**17**	**0.01011**	**(25.99201 32.00619)**
16	2	15360	RK1	J_K	11	0.59522	(26.01895 31.40508)
16	2	15360	RK1	J_S	10	0.34118	(25.9973 31.65883)
16	**2**	**15360**	**RK2**	J_C	**18**	**0.00283**	**(25.99768 32.00162)**
16	2	15360	RK2	J_K	21	0.76624	(26.04455 31.23506)
16	2	15360	RK2	J_S	11	0.44638	(25.99382 31.55366)

Table 6.1.: Optimal Control Optimization results: detection of the origin point s_m of the initial disturbance for the LOH1 equation. We compare three different cost functions: J_C which computes the cost over the complete domain but only at the final time T=1. The sensor cost function J_K computes the cost only over the top boundary. The stabilized sensor cost function J_S which adds the integral over the top boundary at the final time as a stabilization term.

Part V.

Conclusion

7. Conclusion

We looked at two different approaches for computing the adjoint and the gradient. The *Differentiate-then-Discretize* Approach requires the derivation of the analytical adjoint equation. For the simple advection equation example, the derivation is still relatively simple. However, deriving the the Lagrangian function with respect to the state vector can quickly become quite complex for larger equation systems as for example the elastic wave equation. Not for every PDE equation it is possible to derive an adjoint equation. We implemented this approach as a prototype adjoint solver for the advection equation in TerraDG [3]. We saw that the gradient computation requires two additional forward solves and therefore is quite costly. The next step would be to optimize the adjoint computation using the check pointing algorithm suggested in [29]. This would reduce the memory requirements and allow the computation of larger systems and longer simulation times.

We then looked into the *Discretize-then-Differentiate* ansatz, where Automatic Differentiation is used to compute the gradient of a implemented forward solver. We decided to first test the Julia ForwardDiff package [31] in a second prototype adjoint solver in TerraDG. The AD package is quite easy to use but perhaps not the most efficient. For example, it would be interesting to compare the performance to a package using reverse AD. In general, reverse AD is more efficient if the input dimension is larger than the output dimension [30]. When computing the gradient for a large number of parameters, this might become more efficient. The implemented AD adjoint solver prototype could also not completely be tested as the adjoint becomes zero when switching to higher order DG-elements. It is not quite clear what causes this problem. A suspicion, that could how ever not jet be proven, is that the volume term somehow is not correctly differentiated. As shown in the Chapter Tests, the adjoint solution and gradient can still be computed but with a high approximation error. Further tests have to be done to resolve the error for higher order DG and then also do a comparison between first order and higher order gradient computations.

The second goal was to test the dolfin-adjoint package, which uses the finite element solver FEniCS [6], [4]. The software packages we completely new to me and uses a very different approach to the standard numerical PDE solver. With the python interface of FEniCS one can directly implement the weak formulation of a forward model using UFL [19]. If one has previously not worked with such formulations but rather with the standard discrete numerical solvers setups, using the UFL language requires some re-accustoming. With a large number of FEniCS examples and tutorials available, it is quite simple to implement a basic PDE model. The focus of most tutorials are however more elliptic PDE equations. Also the implementation of boundary conditions can be quite different.

As an example optimal control problem we tried determining the origin of the initial disturbance in the LOH1 equation [14]. This showed that the FEniCS and dolfin-adjoint model work quite well. The origin can be detected, even when using a cost functional that computes the cost only at the top boundary over the complete time domain. In general the FEniCS code still has a lot of room for improvement, especially run time optimizations. For finer meshes in combination with higher order DG elements, the FEniCS solvers are still too slow to be able to use them in an optimization routine. This of course would be the next step for improving the solvers. For example, if one can avoiding the assembly of the system matrix in every time loop, this would decrease the forward computation time and with that also the backward solver speed [21].

Bibliography

[1] M. D. Gunzburger, *Perspectives in flow control and optimization*, ser. Advances in design and control. Society for Industrial and Applied Mathematics.

[2] P. E. Farrell. Mathematical background: adjoints and their applications. [Online]. Available: http://www.dolfin-adjoint.org/en/latest/documentation/maths/index.html#dolfin-adjoint-mathematical-background

[3] L. Krenz. Terradg. [Online]. Available: https://github.com/krenzland/TerraDG.jl

[4] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, no. 100, 2015.

[5] A. Logg, K.-A. Mardal, G. N. Wells *et al.*, *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[6] S. K. Mitusch, S. W. Funke, and J. S. Dokken, "dolfin-adjoint 2018.1: automated adjoints for fenics and firedrake," *Journal of Open Source Software*, vol. 4, no. 38, p. 1292, 2019. [Online]. Available: https://doi.org/10.21105/joss.01292

[7] F. S. W. Farrell, Patrick E. Mathematical background: adjoints and their applications. [Online]. Available: http://www.dolfin-adjoint.org/en/latest/documentation/maths/index.html#dolfin-adjoint-mathematical-background

[8] R. J. LeVeque, *Finite volume methods for hyperbolic problems*, ser. Cambridge texts in applied mathematics. Cambridge University Press.

[9] P. Houston and N. Sime, "Automatic symbolic computation for discontinuous galerkin finite element methods." [Online]. Available: http://arxiv.org/abs/1804.02338

[10] D. Corrigan, L. Fullard, and T. Lynch, "Advection problems with spatially varying velocity fields: 1d and 2d analytical and numerical solutions," vol. 146, no. 8, p. 04020053, publisher: American Society of Civil Engineers. [Online]. Available: https://ascelibrary.org/doi/abs/10.1061/%28ASCE%29HY.1943-7900.0001782

[11] A. Fichtner, *Full Seismic Waveform Modelling and Inversion*.

[12] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas, "A high-order discontinuous galerkin method for wave propagation through coupled elasticacoustic media," vol. 229, no. 24, pp. 9373–9396. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0021999110005024

[13] L. Krischer. Seismo-live. [Online]. Available:

[14] SISMOWINE. [Online]. Available: http://www.sismowine.org/index.html?page=model&mset=WP

[15] A. C. Duy, "An introduction to gradient computation by the discrete adjoint method," p. 27.

[16] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, google-Books-ID: VL51G5JYYAYC.

[17] A. Logg and G. N. Wells, "Dolfin: Automated finite element computing," *ACM Transactions on Mathematical Software*, vol. 37, no. 2, 2010.

[18] A. Logg, G. N. Wells, and J. Hake, *DOLFIN: a C++/Python Finite Element Library*. Springer, 2012, ch. 10.

[19] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Transactions on Mathematical Software*, vol. 40, no. 2, 2014.

[20] M. S. Alnæs, *UFL: a Finite Element Form Language*. Springer, 2012, ch. 17.

[21] H. P. Langtangen and A. Logg, "Solving PDEs in python the FEniCS tutorial volume i," p. 153.

[22] Kamensky. Lax-friedrichs flux for advection equation. [Online]. Available: https://fenicsproject.discourse.group/t/lax-friedrichs-flux-for-advection-equation/4647

[23] N.U. Form language unified form language (UFL) 2021.1.0 documentation. [Online]. Available:

[24] J. Hake. fenics-project / dolfin / python / demo / undocumented / dg-advection-diffusion / demo_dg-advection-diffusion.py bitbucket. [Online]. Available:

[25] N.U. DG advection equation with upwinding firedrake documentation. [Online]. Available:

[26] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes," vol. 77, no. 2, pp. 439–471. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0021999188901775

[27] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[28] S. Vandekerckhove. Time-dependent optimal control of the linear scalar wave equation. [Online]. Available: http://www.dolfin-adjoint.org/en/latest/documentation/maths/index.html#dolfin-adjoint-mathematical-background

[29] A. Griewank and A. Walther, "Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation," vol. 26, no. 1, pp. 19–45. [Online]. Available: https://doi.org/10.1145/347837.347846

[30] N.U. Automatic differentiation. [Online]. Available:

[31] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016. [Online]. Available: https://arxiv.org/abs/1607.07892