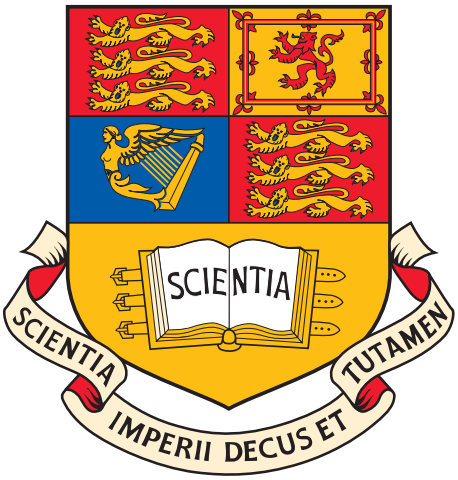
Imperial College London
Department of Earth Science & Engineering

The automation of PDE-constrained optimisation and its applications

Simon W. Funke

October 2012



Supervised by:
Dr. Matthew D. Piggott
Dr. Patrick E. Farrell
Dr. Peter A. Allison
Dr. Gerard J. Gorman

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computational Physics
of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Simon W. Funke

Abstract

This thesis is concerned with the automation of solving optimisation problems constrained by partial differential equations (PDEs). Gradient-based optimisation algorithms are the key to solve optimisation problems of practical interest. The required derivatives can be efficiently computed with the adjoint approach. However, current methods for the development of adjoint models often require a significant amount of effort and expertise, in particular for non-linear time-dependent problems.

This work presents a new high-level reinterpretation of algorithmic differentiation to develop adjoint models. This reinterpretation considers the discrete system as a sequence of equation solves. Applying this approach to a general finite-element framework results in an automatic and robust way of deriving and solving adjoint models. This drastically reduces the development effort compared to traditional methods.

Based on this result, a new framework for rapidly defining and solving optimisation problems constrained by PDEs is developed. The user specifies the discrete optimisation problem in a compact high-level language that resembles the mathematical structure of the underlying system. All remaining steps, including parameter updates, PDE solves and derivative computations, are performed without user intervention. The framework can be applied to a wide range of governing PDEs, and interfaces to various gradient-free and gradient-based optimisation algorithms.

The capabilities of this framework are demonstrated through the application to two PDE-constrained optimisation problems. The first is concerned with the optimal layout of turbines in tidal stream farms; this optimisation problem is one of the main challenges facing the marine renewable energy

industry. The second application applies data assimilation to reconstruct the profile of tsunami waves based on inundation observations. This provides the first step towards the general reconstruction of tsunami signals from satellite information.

Contents

Acknowledgements **xi**

1. Introduction to PDE-constrained optimisation and the adjoint equations **1**

1.1. Introduction 1
1.1.1. From simulation to optimisation 1
1.1.2. Computing derivatives of models 3
1.1.3. Contributions of this thesis 4
1.1.4. Mathematical notation and assumptions 6
1.2. Techniques for computing derivatives 7
1.2.1. Tangent linear approach 9
1.2.2. Adjoint approach 10
1.2.3. Alternative approaches 10
1.2.4. Higher-order derivatives 11
1.3. Developing adjoint models 12
1.3.1. Adjoint of the continuous equations 12
1.3.2. Computing derivatives with algorithmic differentation 16
1.3.3. Adjoint of the discretised equations 20
1.4. Summary and overview 22

2. A library for developing discrete adjoint models **25**

2.1. Introduction 26
2.2. Motivation 28

2.3. The fundamental abstraction 30
2.3.1. Examples 31
2.3.1.1. Diffusion equation 31
2.3.1.2. Burgers' equation 31
2.3.1.3. Time-dependent diffusion equation with exponential source term 32
2.4. The discrete adjoint system 33
2.4.1. Derivation 33
2.4.2. Examples 34
2.4.2.1. Adjoint of the diffusion equation 34
2.4.2.2. Adjoint of the Burgers' equation 34
2.5. Using libadjoint 38
2.5.1. Annotation 38
2.5.2. Recording the forward solutions 39
2.5.3. Function callbacks 40
2.5.3.1. Operator callbacks 40
2.5.3.2. Callbacks for algebraic operations 44
2.5.4. Solving the adjoint system 44
2.5.5. Callback generation with algorithmic differentiation . 45
2.5.6. Verification and debugging 46
2.5.7. Error handling 49
2.6. Description of the core algorithms 50
2.6.1. Deriving the discrete adjoint equations 50
2.6.2. Optimal checkpointing 51
2.6.2.1. Binomial checkpointing 54
2.6.2.2. Determination of required forward solutions in a checkpoint 56
2.7. Examples 58
2.7.1. Burgers' model 58
2.7.2. Shallow water model 62
2.7.2.1. Application to an idealised data assimilation problem 64
2.8. Summary 68

3. Automated generation of adjoint models in FEniCS 71
3.1. Introduction 72

3.2. The FEniCS system 74
3.3. Applying libadjoint to DOLFIN 74
3.3.1. Annotation 75
3.3.2. Recording the forward solutions 76
3.3.3. Callbacks 77
3.3.4. User interface 78
3.4. Implementation 79
3.4.1. Boundary conditions 79
3.4.2. Parallelism 81
3.4.3. Limitations 81
3.5. Examples 82
3.5.1. Burgers' equation 82
3.5.2. Navier-Stokes equations 84
3.5.3. Gray-Scott equations 86
3.6. Summary 89

4. A framework for PDE-constrained optimisation 91
4.1. Introduction 92
4.2. The reduced optimisation problem 93
4.3. Optimisation algorithms 96
4.3.1. Broyden-Fletcher-Goldfarb-Shanno method 96
4.3.2. Sequential quadratic programming 100
4.4. The optimisation framework 101
4.4.1. User interface 101
4.4.2. Implementation 103
4.5. Examples 104
4.5.1. Distributed control of the heat equation 105
4.5.2. Distributed control of a non-linear, time-dependent PDE 108
4.6. Verification 110
4.6.1. Smooth control 112
4.6.2. Bang-bang control 113
4.7. Summary 114

5. Layout optimisation of marine turbine arrays 117
5.1. Introduction 119

5.2. Current methods in wind farm layout optimisation 121
5.3. Mathematical formulation as a PDE-constrained optimisation problem 123
5.3.1. The governing PDE 123
5.3.2. The turbine parameterisation 124
5.3.3. The functional of interest 125
5.3.4. Box and inequality constraints 126
5.4. Numerical setup 126
5.4.1. Optimisation algorithm 126
5.4.2. Spatial discretisation 127
5.4.3. Temporal discretisation 128
5.4.4. Gradient computation 128
5.5. Verification 130
5.5.1. Verification of the forward model 130
5.5.2. Verification of the gradient computation 131
5.6. Examples 132
5.6.1. A single turbine 134
5.6.2. Scenario 1 135
5.6.3. Scenario 2 136
5.6.4. Scenario 3 140
5.7. Conclusions 140

6. Data assimilation for the shallow water equations with wetting and drying 143
6.1. Introduction 145
6.2. Shallow water model with wetting and drying 146
6.2.1. Continuous formulation 146
6.2.2. Spatial discretisation 148
6.2.3. Temporal discretisation 151
6.2.4. Verification 152
6.2.5. Volume conservation 155
6.2.6. Validation 156
6.3. The optimisation problem 156
6.3.1. Mathematical formulation as a PDE-constrained optimisation problem 156
6.3.2. Adjoint model implementation 157

6.4. Numerical examples 158
6.4.1. Wave profile reconstruction on a sloping beach 159
6.4.1.1. Sinusoidal Dirichlet boundary 160
6.4.1.2. Composed sinusoidal Dirichlet boundary . . 162
6.4.2. Reconstruction of the Hokkaido-Nansei-Oki tsunami wave profile 165
6.5. Summary 166

7. Summary and future work **169**
7.1. Summary 169
7.2. Future work 170
7.2.1. Application of libadjoint to other high-level frameworks 170
7.2.2. Automated continuous adjoint approach in FEniCS . 170
7.2.3. Using the automated adjoint derivation and the PDE-constrained optimisation framework 171
7.2.4. Support for multilevel methods for the PDE-constrained optimisation framework 172
7.2.5. Support for solving optimisation problems in the unreduced form 172
7.2.6. Extensions of the optimal positioning of tidal turbines 172
7.2.7. Adjoint model of the Navier-Stokes equations with free-surface and wetting and drying 173

A. Adjoint derivation of the continuous Burgers equation **175**

B. Example usage of algorithmic differentiation **179**

C. Adjoint derivation of the shallow water equations **183**

References **187**

To my family

Acknowledgements

I thank my supervisors for the support during the development of this work. Dr Matthew Piggott was always very supportive and generous. I am grateful for his valuable advice and for giving me the opportunity to travel and participate in many conferences. Much of this research has been done in collaboration with Dr Patrick Farrell. He has not only been a great colleague and supervisor, but also became a good friend. His excellent ideas and invaluable contributions have greatly influenced the outcome of this thesis. I would like to thank Dr Peter Allison and Dr Gerard Gorman for their encouraging advice which they offered so readily.

Furthermore, I would like to thank Dr David Ham who was always willing to discuss my research and give insightful advice. I thank Dr Marie Rognes for the invitation to the Simula Research Laboratory in Oslo. With Drs Farrell, Ham and Rognes emerged a fruitful collaboration which has formed the basis of this thesis. I would like to thank Prof. Christopher Pain for the support and help on the wetting and drying work. I also thank Prof. Zhu Jiang from the Institute of Atmospheric Physics, Beijing for his kind invitation to join the group research trip.

Many thanks go to my colleagues of the Applied Modelling and Computation Group, who have been constantly supportive and I am very grateful for the good working environment they provided. I would like to thank Stephan Kramer for the valuable advice and Dr Jon Hill, Florian Rathgeber, Frank Milthaler, Sam Parkinson, Simon Leo Mouradian, Dave Robinson and Mark Goffin for proofreading parts of this thesis. Thanks also to Dr Tim Greaves and Surinder Singh Dio for making sure that the computer equipment was working.

I gratefully acknowledge the support of the Grantham Institute for Climate Change and the Fujitsu CASE studentship, which funded me during the Ph.D. and provided complete freedom in my research. In addition, I acknowledge the Imperial College Trust for providing conference funding.

I would have not been able to perform this work without the love and support of my family. The numerous visits and mails from my mother, father and my sisters, Leonie and Baucis, gave me the encouragement and support to finish this work. My father proofread this thesis with his enthusiasm, careful eye for detail and his mathematical mind. Finally, a very special thanks goes to my girlfriend, Kirsty. She always stood behind me, provided a stabilising force and helped me in difficult decisions. I cannot thank her enough for everything she has done and endured during these years.

Chapter 1

Introduction to PDE-constrained optimisation and the adjoint equations

Contents

1.1. Introduction 1
1.2. Techniques for computing derivatives 7
1.3. Developing adjoint models 12
1.4. Summary and overview 22

1.1. Introduction

1.1.1. From simulation to optimisation

The use of computational models for simulating physical processes is a powerful complement to physical experiments. Firstly, they allow simulations that are impossible to perform in an experimental approach, such as climate physics or the dynamics of black holes and galaxies. Secondly, computational models can provide information that goes beyond what is observable by experiments, such as derivatives of the simulation result with respect to parameters. This thesis focuses on the computation of these derivatives

and their use in optimisation subject to computational models. The latter is motivated by the fact that many questions in science and engineering can be formulated as such optimisation problems.

In engineering, the need for optimisation emerges regularly when seeking improved designs. A classical example is concerned with a key element in aeronautical engineering:

> What is the optimal shape of an aerofoil?

In the pioneering work of Jameson (1988), this question is considered as an optimisation problem governed by the Euler equations for compressible flow. A similar design problem is investigated in chapter 5 of this thesis. It considers the situation where a large number of tidal turbines are to be deployed in a tidal stream. The investigated optimisation problem is:

> What are the optimal positions of the tidal turbines?

A common task in geosciences is to determine unknown parameters such that the computer models best reproduces existing measurements (Ghil and Malanotte-Rizzoli, 1991). For example, satellite images provide detailed surface information about the ocean, but in general little is known about its interior (Stammer et al., 2000). The ECCO2 project aims to incorporate as much observations as possible to create an accurate description of the time-evolving state of the ocean (Menemenlis et al., 2008). Here, the problem formulation is:

> What is the state of the ocean at the beginning of the observation interval that minimises the difference between simulation and measurements?

Chapter 6 of this thesis considers a related data assimilation problem where satellite observations of a tsunami inundation are available. The question here is:

> Is it possible to reconstruct the profile of the tsunami wave from the observed inundation pattern?

Despite the variety of the considered tasks, they can all be formulated as optimisation problems constrained by partial differential equations (PDEs)

of the form:

$$\begin{aligned} &\min_{u,m} \; J(u,m) \\ &\text{subject to } F(u,m) = 0, \end{aligned} \tag{1.1}$$

where m contains the optimisation parameters, $J(u,m) \in \mathbb{R}$ is the functional of interest that is to be minimised, and $F(u,m) = 0$ represents a PDE parameterised by m with solution u.

As an example, reconsider the optimal design of the aerofoil. In this case, the parameter m contains the parameterised shape of the aerofoil, for example the coefficients of its Bézier curve. The physics are described by the Euler equations, which yield the pressure and velocity for a given aerofoil design. Finally, the functional of interest J computes the performance of the aerofoil, for example by evaluating the drag-lift ratio. The issue that the performance is to be maximised but formulation (1.1) describes a minimisation problem is resolved by minimising the negative of J.

The presented examples have the common feature that the solution of the governing PDE is computationally demanding and that a large number of parameters need to be optimised. This makes solving these optimisation problems difficult and it became only recently tractable with the increase of computational power and the availability of efficient algorithms. In particular, a key role in this progress is the development of new gradient-based optimisation algorithms, such as Quasi-Newton methods (Nocedal and Wright, 1999, §6), and fast gradient computations with the adjoint approach (Errico, 1997).

1.1.2. Computing derivatives of models

Gradient-based optimisation algorithms play a central role in solving PDE-based optimisation problems in practice. One difficulty in their application is that they require the derivative of the model output with respect to its input parameters. Table 1.1 lists some common techniques for computing this derivative information, together with an estimate of their computational expense. For the examples stated above and in many other applications, the number of input parameters is large, typically ranging from 10^2 (in the tidal turbine layout problem) up to 10^9 (in the ECCO2 example, where the parameter describes the initial state of the ocean (Stammer and Wunsch, 1999)) and more. At the same time, the number of output parameters can

Method	Complexity (PDE solves)
Finite difference approximation	$\mathcal{O}(i)$
Complex step approach	$\mathcal{O}(i)$
Tangent linear approach	$\mathcal{O}(i)$
Adjoint approach	$\mathcal{O}(o)$

Table 1.1.: Comparison of the required number of PDE solves for computing the derivative of a computer model with i input and o output parameters. The constant in front of the $\mathcal{O}$ operators is usually one. The PDE solves are typically the principle computational cost factor, and hence provide a good estimate for the total expense. In the given examples, $i = 10^2, \ldots, 10^9$ and $o = 1$ and hence the adjoint approach is by far the most efficient method.

often be reduced to a single value of interest. For the examples above, these are the drag-lift ratio of the aerofoil, the norm of the difference between model and measurements and the power output of the tidal turbines. More generally, this is the case for any problem written in the form (1.1), since the functional of interest J yields a single value. Examining table 1.1 reveals that for such problems, the adjoint approach is the only practical option since the number of PDE solves is independent of the number of input parameters.

Despite the advantages of the adjoint approach, its implementation is regarded as difficult and often combined with extensive development effort, even with the progress in algorithmic differentiation that aims to automate this process. In his recent book, Naumann (2012) states:

> [T]he automatic generation of optimal (in terms of robustness and efficiency) adjoint versions of large-scale simulation code is one of the great open challenges in the field of High-Performance Scientific Computing.

1.1.3. Contributions of this thesis

The first chapters of this thesis present novel techniques for the automated derivation of adjoint models. By using a high-level reformulation of algorithmic differentiation, an automatic and robust way for deriving efficient adjoint models from models written in a high-level finite-element framework is obtained (chapters 2 and 3). Based on this, a framework for PDE-

constrained optimisation is developed, that is novel in terms of generality (both linear and non-linear, and both steady-state and time-dependent PDEs are supported) and its degree to which it is automated (chapter 4). This, together with a high-level language for the problem specification, drastically reduces the effort to solve PDE-constrained optimisation. The framework is applied to two applications for which adjoint-based optimisation has not been used before. Firstly, the optimal positioning of tidal turbines in a tidal stream (chapter 5), which results in new layout designs that have not yet been considered in the literature. Secondly, the application of data assimilation to wetting and drying processes in shallow water (chapter 6), which also has not been studied before. For this thesis, forward and adjoint models of the heat equation, the Burgers' equation, the Navier-Stokes equations, the Gray-Scott equations and the shallow water equations with a wetting and drying extension have been developed, many of which have been used for PDE-constrained optimisation.

The work in this thesis was partly undertaken with collaborators. The library libadjoint presented in chapter 2 was developed in joint work with P.E. Farrell and D.A. Ham. I was centrally involved in the whole design and development process and contributed large parts of the code, in particular to the core data structures and algorithms, the debugging tools. In addition, I was the sole developer of the checkpointing functionality for balancing the computation and storage cost. The integration of DOLFIN with libadjoint presented in chapter 3 is joint work with P.E. Farrell, D.A. Ham and M.E. Rognes. Again, I was centrally involved in the whole design and development process. Large parts of the code such as the Python interface to libadjoint, and the process of annotating and manipulating the equations solved, were performed in pair programming with P.E. Farrell. I was the sole author of major sections of dolfin-adjoint, including the checkpointing support and the optimisation framework presented in chapter 4. For all other work presented in this thesis, I am the sole contributor, in particular the turbine layout optimisation (chapter 5) and the application of data assimilation to wetting and drying processes (chapter 6).

A publication based on the material in chapter 3, the integration of libadjoint in DOLFIN, has been accepted (Farrell et al., 2012). Furthermore, a wetting and drying scheme for the Navier-Stokes equations with free-surface has been developed and published as part of this Ph.D. (Funke

et al., 2011). This scheme will be briefly discussed in section 7.2.7. Publications based on the remaining chapters are in progress.

1.1.4. Mathematical notation and assumptions

The adjoint approach is based on two components: the forward model and the functional of interest. In this thesis, the *forward model* is a (continuous or discretised) system of PDEs written in the abstract form:

$$F(u, m) = 0 \in Z, \tag{1.2}$$

where F is the PDE operator that depends on the parameter $m \in M$ and yields the solution $u \in U$ in suitable Banach spaces M, U and Z. The parameter m might be a source term, initial or boundary conditions, the geometry of the domain, or a coefficient that appears in the PDE. Although the adjoint system itself is independent on m, its introduction is useful for the following derivation and applications. In cases where the choice of the parameter m is irrelevant, it is sometimes neglected and the forward model is simply written as a function of u, i.e. $F(u) = 0$.

The second ingredient is a scalar-valued *functional of interest*, which maps the PDE-solution and the parameter to a value of interest:

$$J(u, m) \in \mathbb{R}. \tag{1.3}$$

Similarly, the dependency on the parameter m is sometimes neglected and the functional of interest simply written as $J(u)$.

Throughout this thesis, it is assumed that the functional J and the PDE-operator F are continuously Fréchet differentiable. Fréchet differentiability is a generalisation of the differentiability from functions in $\mathbb{R}^n$ to Banach spaces:

Definition 1. *Let $f : A \subset X \to Y$ be an operator with Banach spaces X, Y and $A \neq \emptyset$ open. f is called Fréchet differentiable if for all $x \in A$ there exists a bounded linear operator $Df(x) : X \to Y$ such that:*

$$\lim_{\|h\|_X \to 0} \frac{\|f(x+h) - f(x) - Df(x)h\|_Y}{\|h\|_X} = 0.$$

f is continuously Fréchet differentiable if the function $x \to Df(x)$ is continuous.

In addition, it is assumed that the forward model (1.2) yields a unique solution u for any $m \in M$. This allows the definition of a solution operator $u(m)$, which maps the parameter m to the associated solution of the forward model. Of course, for many PDEs the explicit form of $u(m)$ is not known, and instead a program is used to compute an approximation.

Substituting the solution operator $u(m)$ into the functional of interest (1.3), yields the *reduced functional*:

$$\tilde{J}(m) \coloneqq J\left(u(m), m\right). \tag{1.4}$$

An evaluation of the reduced functional of interest consists of first solving the forward model to obtain u and then evaluating J as in (1.3).

It is also assumed that the linearised PDE operator $\partial F(u(m), m)/\partial u$ is invertible for any $m \in M$. The implicit function theorem shows that the solution operator $u(m)$ is then continuously Fréchet differentiable (Hinze et al., 2009, §1.4.2). A detailed discussion about the mathematical requirements for the adjoint approach can be found in (Hinze et al., 2009, §1.6).

A final note on the notation of partial and total derivatives. If m consists of multiple entries arranged as a row vector $m = (m_1, m_2, \dots)$, then the total derivative is thought of as a row vector of the form:

$$\frac{\mathrm{d}J}{\mathrm{d}m} = \left(\frac{\mathrm{d}J}{\mathrm{d}m_1}, \frac{\mathrm{d}J}{\mathrm{d}m_2}, \dots\right).$$

Similarly, the partial derivative is in this case:

$$\frac{\partial J}{\partial m} = \left(\frac{\partial J}{\partial m_1}, \frac{\partial J}{\partial m_2}, \dots\right).$$

1.2. Techniques for computing derivatives

This section derives the equations for computing the (total) derivative of the functional with respect to the parameters, i.e. $\mathrm{d}J/\mathrm{d}m$. Taking the derivative of equation (1.4) and applying the chain rule on its right hand side yields:

$$\frac{\mathrm{d}\tilde{J}}{\mathrm{d}m} = \frac{\mathrm{d}J}{\mathrm{d}m} = \frac{\partial J}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial J}{\partial m}. \tag{1.5}$$

If equation (1.5) was used directly to compute the functional gradient, all three terms on the right hand side would be needed. For the terms involving J this is straightforward: the functional of interest is usually given as an explicit analytical formula from which the derivatives can be calculated. However, $u(m)$ is only given implicitly through the solution operator and therefore $\mathrm{d}u/\mathrm{d}m$ is not directly accessible.

One way to compute $\mathrm{d}u/\mathrm{d}m$ is obtained by taking the derivative of the forward model (1.2) with respect to m:

$$\frac{\partial F}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial F}{\partial m} = 0. \tag{1.6}$$

This equation contains the unknown $\mathrm{d}u/\mathrm{d}m$ and the derivatives of the forward model with respect to u and m. The PDE operator F is always available in explicit form from which the derivatives can be obtained. Since the linearised PDE operator $\partial F/\partial u$ is assumed to be invertible, the following equation for $\mathrm{d}u/\mathrm{d}m$ is obtained:

$$\frac{\mathrm{d}u}{\mathrm{d}m} = -\frac{\partial F}{\partial u}^{-1}\frac{\partial F}{\partial m}. \tag{1.7}$$

Substituting equation (1.7) into (1.5) yields an equation for the functional gradient with terms that can all be derived explicitly:

$$\frac{\mathrm{d}J}{\mathrm{d}m} = -\overbrace{\frac{\partial J}{\partial u}\underbrace{\frac{\partial F}{\partial u}^{-1}\frac{\partial F}{\partial m}}_{=-\mu}}^{=\lambda^*} + \frac{\partial J}{\partial m}, \tag{1.8}$$

where the asterisk denotes the Hermitian transpose. The order in which the first term on the right hand side is evaluated leads to two different approaches: one involves solving the *tangent linear model* for μ, and the other one solves the *adjoint model* for λ.

To facilitate the comparison of these two approaches, a dimensional analysis is performed in the following sections. For that, let $|m|$ and $|u|$ denote the (finite) dimensions of the parameter m and the forward model solution u, respectively. Figure 1.1 visualises the dimensions of the linear operators in the right hand side of equation (1.8).

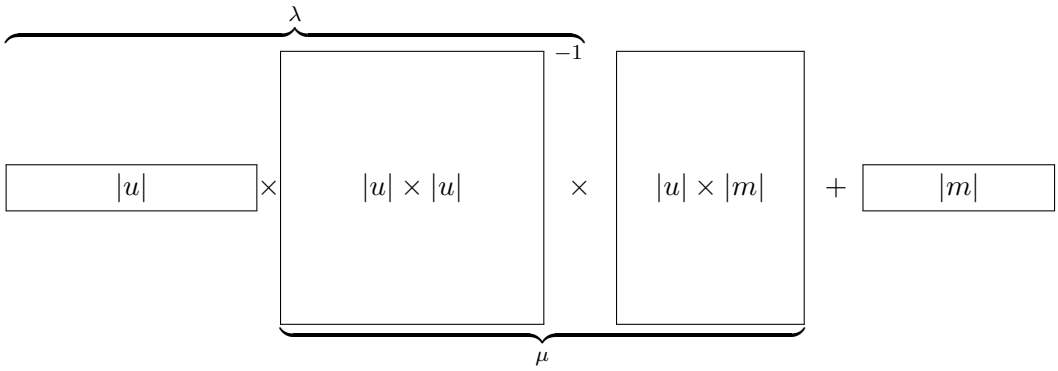


Figure 1.1.: The dimensions of the linear operators in the right hand side of equation (1.8), visualised for a finite-dimensional example. The tangent linear solution μ is a matrix of dimension $|u| \times |m|$ and requires the solution of m linear systems. The adjoint solution λ is a vector of dimension $|u|$ and is obtained with a single linear solve.

1.2.1. Tangent linear approach

The tangent linear approach computes the functional derivative by first solving the tangent linear model for μ:

$$\frac{\partial F}{\partial u}\mu = -\frac{\partial F}{\partial m}, \tag{1.9}$$

and then using equation (1.8) to obtain the sought gradient:

$$\frac{\mathrm{d}J}{\mathrm{d}m} = \frac{\partial J}{\partial u}\mu + \frac{\partial J}{\partial m}.$$

Note that the linearised forward model (1.6) and the tangent linear model (1.9) are equivalent and hence $\mu = \mathrm{d}u/\mathrm{d}m$.

The computational cost of this approach is dominated by the solution of the tangent linear system (1.9). The dimensional comparison in figure 1.1 reveals that the right hand side of this linear system is a matrix with $|m|$ columns, corresponding to the number of parameters. One option to solve this system is to invert the linearised PDE operator $\partial F/\partial u$ explicitly (for example with a LU decomposition) and to perform a matrix-matrix multiplication to compute μ. Alternatively, $|m|$ iterative solves can be performed

for each column in the right hand side of the tangent linear system (1.9). However, in both cases, the computational cost scales linearly with $|m|$.

The next section shows that the adjoint approach leads to a linear system whose dimension is independent of the parameter m.

1.2.2. Adjoint approach

The adjoint approach first solves the adjoint equation:

$$\frac{\partial F}{\partial u}^* \lambda = \frac{\partial J}{\partial u}^* . \tag{1.10}$$

The adjoint solution λ is then used to compute the functional gradient using equation (1.8):

$$\frac{\mathrm{d}J}{\mathrm{d}m} = -\lambda^* \frac{\partial F}{\partial m} + \frac{\partial J}{\partial m}. \tag{1.11}$$

The dimensional analysis in figure 1.1 shows that the right hand side of the adjoint equation (1.10) always consists of a single column vector, corresponding to the scalar-valued functional of interest J. Hence, the computation of the adjoint solution λ requires only one linear solve, independent of the dimension of m. In fact, the adjoint equation (1.10) does not depend on the choice of m at all. Furthermore, since the adjoint system is based on a linearisation of the forward model, its computational cost is expected to be equal or less than that of the forward model. However, in practice the efficient implementation of an adjoint model is often difficult, as will be discussed in the section 1.3.

1.2.3. Alternative approaches

Apart from the tangent linear and adjoint approaches, there are alternative ways to obtain the functional gradient. Two common methods are the finite-difference and the complex-step approaches which will be briefly discussed in this section.

The finite-difference approach approximates the functional gradient in the direction δm by computing the difference quotient. The central-difference formula for example yields:

$$\frac{\mathrm{d}\tilde{J}(m)}{\mathrm{d}m}\delta m = \frac{\tilde{J}(m + h\delta m) - \tilde{J}(m - h\delta m)}{h} + \mathcal{O}(|h|^2) \quad \text{as } h \to 0.$$

The main advantage of the finite-difference approach is its easy implementation: the model can be treated as a black box since only functional evaluations are required. However, the determination of a suitable step length h can be difficult: from a mathematical perspective, h should be chosen as small as possible to improve the approximation, but due to numerical cancellation the smallest suitable h value is bounded by round-off errors (Nocedal and Wright, 1999, pp. 166–169).

The complex-step approach (Lyness and Moler, 1967) avoids this problematic by using complex calculus. It considers the reduced functional in the complex plane and uses the Cauchy-Riemann equations to derive following approximation of the directional derivative:

$$\frac{\mathrm{d}\tilde{J}(m)}{\mathrm{d}m}\delta m = \frac{\mathrm{Im}\left(\tilde{J}(m + ih\delta m)\right)}{h} + \mathcal{O}(|h|^2) \quad \text{as } h \to 0.$$

Since no difference operation is performed, this evaluation is not subject to subtractive cancellation errors. From an implementation perspective the complex-step approach is more intrusive, since the underlying code must be modified to support complex numbers. It has been recently shown that this method is tightly related to the forward mode of algorithmic differentiation described below (Martins et al., 2003).

Both the finite-difference and the complex step approach only yield approximations of directional derivatives. To obtain the full functional gradient, the directional derivatives for all basis vectors in the parameter space must be computed separately. As a consequence the computational complexity increases linearly with the dimension of m. Nevertheless, the finite-difference method is popular due to its straightforward implementation and is a useful verification tool, see section 2.5.6.

1.2.4. Higher-order derivatives

It is possible to compute higher-order derivatives by recursively applying the adjoint or tangent linear approach to itself. A detailed derivation of second-order adjoints can be found in Griewank and Walther (2008, §13.4), Hinze et al. (2009, §1.6.5) and Thevenin and Janiga (2008, §4.6).

One application of second-order information is in the context of optimisation with PDE constraints, for example for design optimisation (Guillaume

and Masmoudi, 1994) and optimal control (Hinze and Kunisch, 2001; Raffard and Tomlin, 2005).

1.3. Developing adjoint models

The implementation of an adjoint model is often complicated and can require years of development time for complex forward models. This section introduces different options for how the adjoint model can be derived and highlights their advantages and problems.

The development of a PDE based forward model can be divided into three stages, as visualised in the top row of figure 1.2. First, it is decided which system of PDEs is to be solved. Then these equations are discretised in space and time. For that, a common choice is to first perform the spatial discretisation using the finite element method (Elman et al., 2005), and then apply a time stepping scheme such as Runge-Kutta methods (Ascher and Petzold, 1998). This separation of space and time discretisation is also known as the method of lines (Schiesser, 1991). The final step of the forward model development is the implementation of the discretised equations as source code.

The adjoint model can be derived at any of these three stages, as shown in figure 1.2. The resulting adjoint system is at the same discretisation stage as the forward model from which it was derived. That is, the continuous adjoint is a continuous PDE, the discrete adjoint is a discretised PDE and the adjoint of the source code is an implementation of this discretised PDE. Interestingly, the diagram in figure 1.2 does not commute. As a result, the adjoint yields different solutions depending on how it was derived. The following sections discuss these options in more detail with a specific emphasis on how the derivation can be automated.

1.3.1. Adjoint of the continuous equations

The continuous adjoint PDE is obtained by deriving the adjoint system of the continuous forward PDE prior to discretisation.

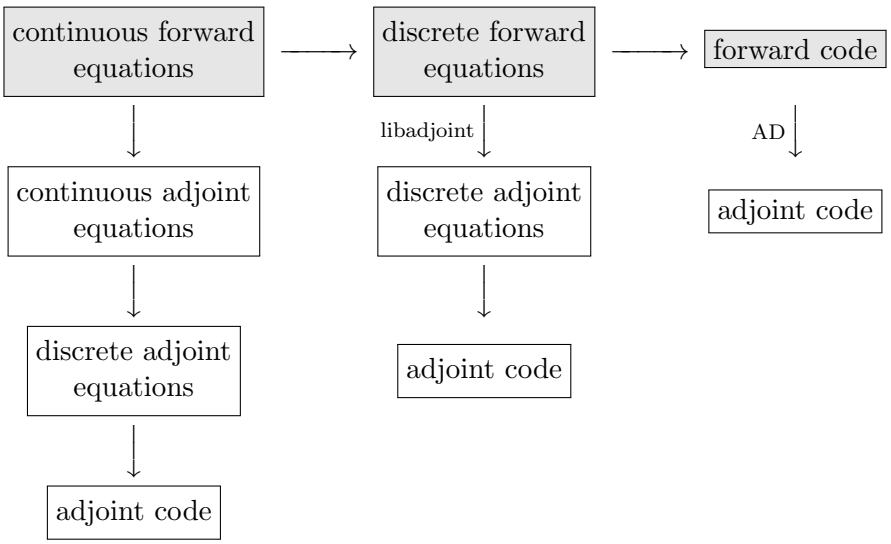


Figure 1.2.: The adjoint system can be derived after any stage of the forward system development (top row). The derivation of the continuous adjoint is usually performed by hand. libadjoint is a library presented in chapter 2 that facilitates the development of the discrete adjoint. Algorithmic differentiation (AD) tools derive the adjoint model from the forward model source implementation. The resulting adjoint code and the adjoint solution depend in general on the stage at which the adjoint system was derived.

As an example, consider the time-dependent viscous Burgers' equation with Dirichlet boundary conditions:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u = 0 \qquad \text{in } \Omega \times (0,T), \tag{1.12a}$$

$$u = g \qquad \text{on } \partial\Omega \times (0,T), \tag{1.12b}$$

$$u = u_0 \qquad \text{for } \Omega \times \{0\}, \tag{1.12c}$$

where $\Omega \subset \mathbb{R}$ defines the spatial and $(0,T)$ the temporal domain for the solution $u : \Omega \times (0,T) \to \mathbb{R}$, $\nu \in \mathbb{R}$ is the viscosity parameter, $g : \partial\Omega \times (0,T) \to \mathbb{R}$ describes the Dirichlet boundary value and $u_0 : \Omega \to \mathbb{R}$ the initial condition. The continuous adjoint of the Burgers' equation with respect to a functional of interest $J(u,m) \in \mathbb{R}$ reads as (see appendix A for a derivation):

$$\begin{aligned} -\frac{\partial \lambda}{\partial t} - (u \cdot \nabla)\lambda + (\nabla u)^* \lambda - \nu \nabla^2 \lambda = \frac{\partial J^*}{\partial u} \quad & \text{in } \Omega \times (0,T), \\ \lambda = \frac{\partial J^*}{\partial u} \quad & \text{on } \partial\Omega \times (0,T), \\ \lambda = \frac{\partial J^*}{\partial u} \quad & \text{for } \Omega \times \{T\}, \end{aligned} \tag{1.13}$$

where ∇u denotes the Jacobian matrix of u and $\lambda : \Omega \times (0,T) \to \mathbb{R}$ is the adjoint solution. This adjoint PDE can be discretised and solved with similar techniques as the forward PDE.

The continuous adjoint of the Burgers' equation (1.13) reveals some general properties about the adjoint of a time-dependent problem: the resulting PDE is linear with initial conditions at the end of the time interval. Consequently it is solved backwards in time. Furthermore, the adjoint PDE depends on the forward solution u, and therefore the forward model must be solved beforehand.

There are a few reasons why the derivation of the continuous level is preferable over its alternatives. Firstly, it allows the independent discretisation of forward and adjoint system. This flexibility can be important if the adjoint solution has different characteristics to the forward solution. Choosing different discretisation schemes and resolutions can increase the numerical accuracy for the same computational cost, see for example Pelletier et al. (2003); Fang et al. (2006).

The second advantage is that discontinuities introduced by numerical methods are unproblematic. A common example are upwind schemes to solve hyperbolic differential equations (Liu and Sandu, 2005), but also algorithms for space and time adaptivity are typically non-differentiable.

Finally, the continuous adjoint plays an important role in shape optimisation as it avoids the issue of computing the derivative of the functional with respect to the mesh node positions (Anderson and Venkatakrishnan, 1999).

However, this flexibility has the caveat that the functional gradient derived from the adjoint solution is generally not the functional gradient of the discrete forward model. An extreme case is presented in Gunzburger (2003, §4.1.2) where the functional gradient computed with the continuous adjoint approach takes the opposite direction to the functional gradient of the discrete forward model. The reason for this inconsistency is that the forward and adjoint models are independently discretised. Hence, while forward and adjoint solutions converge to their continuous solutions, they can be arbitrarily unrelated if discretisation errors dominate. The consequences have been investigated by Griesse and Walther (2004) and Gunzburger (2003, §4.1.2) where the authors showed that this inconsistency can impact the convergence of gradient-based optimisation algorithms.

The manual derivation of the continuous adjoint system can be laborious for complex PDEs and the subsequent discretisation and implementation duplicates the development effort. This can become particularly problematic for models that are still in development: whenever the discrete forward system changes, the adjoint derivation has to be repeated and the adjoint model implementation updated.

An automation of the continuous adjoint approach would need to be able to process the steps involved in the appropriate path in figure 1.2. The first step consists of the derivation of the continuous adjoint PDE from the forward PDE. This derivation can be automated, for example by using a symbolic algebra package such as SAGE (Stein et al., 2011). The second step, the discretisation of the resulting adjoint PDE, is more difficult to automate: the choice of a suitable discretisation scheme for a given PDE usually requires user expertise. Finally, the discretised equations must be passed to a code generation tool that automatically emits the adjoint model source code. While such tools are difficult to develop in general, there exists several

"problem solving environments" for the finite element method, for example FEniCS (Logg et al., 2011), Sundance (Long et al., 2012), GetDP (Dular et al., 1998), deal.II (Bangerth et al., 2007) and Analysa (Bagheri and Scott, 2004). Besides the potential of the continuous approach, the author is not aware of a software package that automates the required steps.

1.3.2. Computing derivatives with algorithmic differentation

An alternative to the continuous adjoint approach is to postpone the derivation until the last stage of the forward model development, see figure 1.2. The resulting derivation operates at a much lower level: the source code.

In this approach, the forward model is considered as a sequence of elementary instructions such as $+, \cdot, \sin$ or $\exp$, for each of which the derivative is known. That is, the reduced functional (1.4) can be written as a sequence of elementary instructions:

$$\tilde{J}(m) = f_p \circ f_{p-1} \circ f_{p-2} \circ \cdots \circ f_1(m). \tag{1.14}$$

Differentiating equation (1.14) and applying the chain rule to its right hand side yields an equation for the functional gradient:

$$\frac{\mathrm{d}\tilde{J}}{\mathrm{d}m} = \frac{\partial f_p(m_{p-1})}{\partial m_{p-1}} \frac{\partial f_{p-1}(m_{p-2})}{\partial m_{p-2}} \cdots \frac{\partial f_1(m)}{\partial m}, \tag{1.15}$$

where m_k is the k'th intermediate solution of equation (1.14), i.e. the result of the first k function compositions. There are two ways in which the right hand side of (1.15) is commonly evaluated. The first one, known as forward accumulation or forward mode, evaluates the terms from right to left. This evaluation involves only derivatives of elementary instructions for which the derivatives are known.

The forward mode is tightly related to the tangent linear model (section 1.2.1): by observing that the reduced functional (1.4) can be written as $\tilde{J}(m) = J(\cdot, m) \circ u(m)$, there exists a p' such that $u(m) = f_{p'} \circ f_{p'-1} \circ \cdots \circ f_1(m)$, and hence $\mathrm{d}f_{p'}/\mathrm{d}m = \mathrm{d}u/\mathrm{d}m = \mu$ appears as an intermediate variable during the forward accumulation. The forward accumulation also adopts the dependency properties of the tangent linear model; in particular the calculation can be performed in line with the forward model without storing any additional intermediate variables.

The second way is to evaluate the right hand side of equation (1.15) from left to right, which is also known as reverse accumulation or reverse mode. This mode adopts the properties of the adjoint approach: in particular the gradient evaluation is performed backwards and all intermediate forward variables $m_i, i = 1, \ldots, p$ must be stored.

In contrast to the continuous adjoint approach, the gradient information obtained with this approach is consistent with the discrete forward model. That is, the computed gradient is the exact derivative of the forward model implementation. While this is generally desirable, in particular in the context of PDE based optimisation, there are cases where the consistent approach leads to an unsuitable discretisation of the continuous adjoint equation. Sirkes and Tziperman (1997) investigated this issue and present an example for which the adjoint solution contains numerical artifacts.

The key advantage of this approach is that the derivation and evaluation of the adjoint model follow a completely prescriptive process, for which all required information is directly accessible from the source code. Consequently, the adjoint model can, in theory, be obtained fully automatically. This idea is implemented in automatic or algorithmic differentiation (AD) tools (Griewank and Walther, 2008), which commonly use one of the following techniques:

Operator overloading. Uses the operator overloading feature of modern computer languages to build a tape of all executed functions and their arguments at runtime. This tape is then differentiated and evaluated to obtain the derivative information. This approach is relatively straightforward to implement but yields less efficient derivative code because no compiler optimisations can be performed on the differentiated code.

Source to source compiler. A typical source-to-source AD tool takes the source code of the forward model as an argument and generates the source code for computing its derivative. Compared to operator overloading tools, source to source compilers tend to produce more efficient code but their implementation is more challenging.

An example application of the source to source AD tool TAPENADE (Hascoët and Pascual, 2004) to a demonstration program can be found in appendix B.

An authoritative survey of the field can be found in Griewank and Walther (2008); Griewank (2003); Rall and Corliss (1996) and Bücker et al. (2006).

The main advantage of these low-level AD tools is that they can be applied directly to an existing forward model implementation. This is a major feature for complex models for which the manual derivation and implementation of the adjoint equations would be prohibitive. AD tools have been successfully applied to differentiate large models such as the MITgcm general circulation model (Heimbach et al., 2005), the FLUENT CFD code (Bischof et al., 2007), the CICE sea-ice model (Kim et al., 2006), and the WRF weather forecasting model (Xiao et al., 2008).

However, their application have several limitations that need to be taken into consideration. Since these AD tools operate on the source code level they are typically designed for one specific programming language. In practice this raises two problems. Firstly, often only a subset of the programming language is supported, requiring the model developer to avoid advanced language features. For example, TAPENADE, a leading source to source AD tool for C and Fortran, prohibits the usage of dynamic memory allocations and pointer analysis in Fortran 95 in reverse mode (Hascoët and Pascual, 2004, §10.4). Secondly, it complicates the usage of multiple programming languages within the same model. A separate AD tool must be applied to the different parts of the code and manual intervention is needed to organise the differentiation between them. In particular, this can prohibit the use of external numerical libraries for which no differentiated version is available.

The result of these restrictions can be seen in MITgcm, the flagship application for both the TAF (Giering and Kaminski, 2003) and OpenAD (Utke et al., 2008) source to source AD tools. Firstly, it is written in Fortran 77, which is fairly straightforward to parse. Secondly, the numerics are mostly explicit: the implicit step is self-adjoint (Heimbach et al., 2005), which means that no derivative code needs to be generated for the equation solve. Finally, the model has no hard dependencies on any external libraries, all of the core numerical calculations are performed within the model itself.

Another issue for low-level AD tools is that they require significant expertise in both their usage and the forward model in order to derive efficient adjoint code. This is best explained by means of a simple example: consider a forward model that solves one non-linear system using the Newton method up to machine precision. Since the adjoint system is linear, an efficient ad-

joint implementation performs only one linear solve. Instead, without user intervention the AD tool differentiates every single Newton iteration. The result is that the adjoint model solves as many linear solves as there are Newton iterations in the forward model. This issue can be avoided by annotating the forward model with AD tool-specific directives to supply the necessary information. Heimbach et al. (2010) states that "work is thus required initially to make the model amenable to efficient adjoint code generation for a given AD tool. This part of the adjoint code generation is not automatic and can be substantial for legacy code, in particular if the code is badly modularized and contains many irreducible control flows". The adjoint code produced by low-level AD tools is typically 3 to 30 times slower than the forward simulation (Naumann, 2012).

Finally, advanced features, such as the use of checkpoints to balance storage and computation costs that will be described in section 2.6.2, are often only implemented in mature AD packages and usually require further directives from the model developer (Kowarz and Walther, 2006; Hascoët and Araya-Polo, 2006). Also, support for parallel programs is still an active field of research (Hovland, 1997; Utke et al., 2009; Förster et al., 2011).

In summary, the adjoint derivation on the implementation level has the advantage that it can be automated if an AD tool is available. However, in practice multiple caveats appear and their application often requires extensive user intervention and expertise. Vidard et al. (2008) states his experiences of applying a low-level AD tool to the NEMO ocean model as: "Even for this simplified configuration, however, substantial human intervention and additional work was required to obtain a usable product from the raw [AD]-generated code ... [The] memory management and CPU performance of the raw code were rather poor. ... From that experience it has been decided to go toward the hand-coding approach". These problems arise from the low abstraction on which the algorithmic differentiation is applied. By reasoning on the source code level, the differentiation tools have no high-level information about the forward model and hence have to handle low-level implementation details such as parallelism.

1.3.3. Adjoint of the discretised equations

Motivated by the drawbacks of the previous two methodologies, this section and the following chapter attempt to circumvent their problems by choosing a different level of abstraction. On the one hand, it should be low enough such that the derivation of the adjoint system is a purely prescriptive process that can be automated. This is the case once the discretisation type has been chosen, along with the discrete function spaces for all variables. On the other hand, the abstraction should be as high as possible to avoid the implementation details of the forward model. From figure 1.2 it can be seen that deriving the adjoint from the discretised forward model matches these conditions.

To illustrate the approach reconsider the time-dependent Burgers' equation (1.12). Discretising with the finite element method in space and the forward Euler method in time, and linearising the non-linear advective term around the solution at the previous time level, yields the iteration:

$$\begin{aligned} u_0 &= g, \\ \frac{1}{\Delta t} M u_{n+1} - \frac{1}{\Delta t} M u_n + V(u_n) u_n + D u_n &= 0 \quad \text{for } n \in 0, \dots, N, \end{aligned} \tag{1.16}$$

where the subscripts indicate the time levels, M is the mass matrix, $-D$ is the discretised diffusion operator, $V(u_n)$ is the advection matrix assembled at the velocity u_n, and N indicates the number of time step iterations. For brevity, define

$$T(\cdot) \coloneqq -\frac{1}{\Delta t} M + V(\cdot) + D. \tag{1.17}$$

The assignments (1.16) can be written as:

$$\begin{pmatrix} I & & & \\ T(u_0) & \frac{1}{\Delta t} M & & \\ & T(u_1) & \frac{1}{\Delta t} M & \\ & & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} g \\ 0 \\ 0 \\ \vdots \end{pmatrix}. \tag{1.18}$$

The lower-triangular form of the matrix encodes the forward temporal flow of information in the equations: the solutions at later time levels depend on the solutions at earlier time levels, but not vice-versa.

Given a functional of interest J, the discrete adjoint equation of the forward model (1.18) is (the derivation is performed in section 2.4.2.2):

$$\begin{pmatrix} I^* & \left(T(u_0) + \frac{\partial V(u_0)}{\partial u_0} u_0\right)^* & & \\ & \frac{1}{\Delta t} M^* & \left(T(u_1) + \frac{\partial V(u_1)}{\partial u_1} u_1\right)^* & \\ & & \frac{1}{\Delta t} M^* & \ddots \\ & & & \ddots \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} \frac{\partial J^*}{\partial u_0} \\ \frac{\partial J^*}{\partial u_1} \\ \frac{\partial J^*}{\partial u_2} \\ \vdots \end{pmatrix}. \tag{1.19}$$

This system recovers many of the properties of the continuous adjoint system (1.13). The adjoint operator is upper-triangular, hence the temporal flow of information in the adjoint system is reversed. Similarly to the continuous version, the discrete adjoint system depends on the forward solutions u_n.

For this example, the discrete adjoint results in the same time discretisation scheme that was used for the forward model. To demonstrate this, the second equation in the adjoint system (1.19) is expanded:

$$\frac{1}{\Delta t} M^* \lambda_1 - \frac{1}{\Delta t} M^* \lambda_2 + V^*(\lambda_2) + D^* \lambda_2 + \left(\frac{\partial V(u_1)}{\partial u_1} u_1\right)^* \lambda_2 = \frac{\partial J}{\partial u_1}^*.$$

It can be seen that the adjoint variable λ_1 occurs only in the discretised time derivative term. Since the adjoint equations are solved backwards in time, this corresponds to a forward Euler method. In general, Sandu (2006) showed the discrete adjoint of explicit and implicit Runge-Kutta methods of any order p, result in a p-th order discretisation of the continuous adjoint. However, depending on the dynamics of the adjoint solution, this discretisation might be only appropriate for the forward model and result in numerical instabilities in the adjoint solution (Sirkes and Tziperman, 1997).

The main advantage of deriving the adjoint from the discretised equations is that the process is completely prescriptive: once the forward model has been discretised, there is a defined procedure on how the associated adjoint model implementation is obtained. This property is the key for chapter 2, where a library is presented that automates this derivation. Another advantage is that, in contrast to deriving the adjoint from the source code, the derivation of discrete adjoint system is separated from its implementation. In particular, the operators in the adjoint equation (1.19) can be written in

multiple programming languages, rely on external libraries and be parallel aware.

Finally, since the adjoint is derived on the discrete level, the resulting gradient information is consistent with the discrete forward model. Many optimisation algorithms such as SQP (section 4.3.2) and BFGS (section 4.3.1) rely on exact derivatives, and can fail to converge if only approximations are provided (Griesse and Walther, 2004); therefore, the consistency of the discrete adjoint approach is an important property for the remaining chapters of this thesis, where the gradients are used in the context of PDE-constrained optimisation.

1.4. Summary and overview

The adjoint approach provides an efficient way to compute derivative information of PDE-based models in cases where the number of output values of interest is larger then the number of input parameters. Three approaches for developing the associated adjoint models have been introduced and their advantages and disadvantages discussed. The continuous approach provides most flexibility to the adjoint model developer, but requires usually a significant amount of development work. Alternatively, the adjoint model can be derived directly from the implementation of the forward model. Algorithmic differentiation tools aim to automate this process, however in practice their applicability can be limited and they often require user intervention. Finally, the discrete adjoint approach derives the adjoint model from the discretised forward equations. This approach has the advantage that the adjoint derivation is performed independently of implementation details.

The remaining chapters are organised as follows. The next chapter introduces a software library that facilitates the development of adjoint models based on the discrete adjoint approach. Chapter 3 applies this library to a high-level finite-element framework, which results in an automatic, robust and efficient way of deriving and implementing adjoint models. This work is extended in chapter 4, in which a framework for rapidly setting up and solving PDE-constrained optimisation problems is developed. Chapter 5 applies this framework to the problem of finding the optimal positions of tidal turbines in a tidal stream. Another application is given in chapter 6, where a tsunami wave profile is reconstructed from observed inundation

pattern. Finally, chapter 7 makes some concluding remarks and presents future work.

Chapter 2

A library for developing discrete adjoint models

Contents

2.1. Introduction 26
2.2. Motivation 28
2.3. The fundamental abstraction 30
2.4. The discrete adjoint system 33
2.5. Using libadjoint 38
2.6. Description of the core algorithms 50
2.7. Examples 58
2.8. Summary 68

A publication based on this chapter is in preparation in collaboration with P.E. Farrell and D.A. Ham. The author was centrally involved in all aspects of the work.

Abstract
This chapter presents a new software library that facilitates the development of adjoint models based on a high-level abstraction. Its fundamental concept is to consider the forward model as a sequence of equation solves. Based on this abstraction, the library builds a symbolic description of the forward model, from which it can automatically derive the symbolic representation of the associated adjoint model. The contained symbolic operators are then linked to implementations of these operators, with which the library can assemble and solve the adjoint equations. The chapter finishes with an application of this library to develop the adjoint of two example models, for which a direct low-level AD approach would be prohibitive.

2.1. Introduction

The numerical core of a PDE based forward model typically consists of a sequence of equation assemblies and solves. However, the actual implementation of this functionality is often filled with machine-specific details, such as memory management, parallel communication and in-/output. As a consequence, the mathematical purpose of the model is interwoven with details on how it is to be implemented on a particular platform.

A direct application of a low-level AD tool, which operates on the implementation level, needs to differentiate through all of these implementation details. This is the reason for many of the practical problems of applying such tools to complex models (see section 1.3.2), as the differentiation of the implementation details can be difficult. A typical example is the support of parallel communication which is a topic of active research in the AD community (Hovland, 1997; Utke et al., 2009; Förster et al., 2011).

Section 1.3.3 described an alternative derivation of the adjoint model by considering the forward model as a sequence of equation solves. The main contribution of this chapter is the development of the library libadjoint, which uses this abstraction to facilitate and partly automate the development of adjoint models. The application of libadjoint to a forward model can roughly be divided into two steps. First, the forward model must be annotated, i.e. libadjoint functions are added to each equation solve in the forward model in order to pass some necessary information to libadjoint.

This information includes the structure of the equation, the variable it solves for and its linear and non-linear dependencies. As a result, libadjoint has a high-level representation of the forward model, also known as a tape. This tape is analogous to the concept of a tape in AD, except that instead of recording individual elementary operations, the units on the tape are whole equation solves. With the tape, libadjoint can derive the high-level representation of the associated adjoint equations; however, the operators in the derived adjoint equations are at that point just purely abstract handles. Therefore, the second development steps consist of providing implementations of all occurring operators. With these, libadjoint can then assemble and solve the adjoint equations.

The advantage of this approach is that libadjoint's representation of the forward equations is separate from implementation details. Consequently, the difficult derivation of the discrete adjoint system can be automated independently of details such as parallelism, external libraries, etc. The implementation of the required operators is usually relatively easy, and can be performed either manually or with the help of an additional AD tool. The library also manages the storage of forward and adjoint variables and supports the use of checkpointing algorithms to balance the storage and computation cost.

Alternatively, libadjoint can be viewed as a high-level AD tool, where the elemental functions are arbitrarily complex and their derivatives and Hermitian transpose must be provided by the user. A related project is CasADi (Andersson et al., 2012), a symbolic framework for solving optimal control problems governed by ordinary differential equations with build-in algorithmic differentiation features. In this framework, the user builds a syntax tree by performing vector and matrix operations, which is then used to evaluate the function and its derivative. Another related project is YAO (Nardi et al., 2009). This project is motivated by finite difference schemes and regards the model as a repetitive execution of similar core functions, such as a computation on each grid point. By duplicating and connecting these modules, the forward and its associated adjoint model can be generated and executed. In contrast to these tools, libadjoint is specifically designed to be applied to existing forward models and focuses on computationally expensive models such as for solving PDEs.

2.2. Motivation

To motivate the abstraction that libadjoint takes, the three options for developing an adjoint model introduced in section 1.3 are evaluated for two existing software packages: Fluidity, a computational fluid dynamics framework currently under active development (Piggott et al., 2008) and the FEniCS system, a collection of software components for automating the solution of PDEs by the finite element method (Logg et al., 2011; Logg, 2007).

The application of the continuous adjoint approach (see section 1.3.1) to Fluidity is complicated by several factors. Fluidity is highly configurable about which and how equations are to be solved. In particular it supports the Navier-Stokes, the Stokes and the shallow water equations with various parametrisations and an arbitrary number of additional tracer equations. The application of the continuous adjoint approach would require the adjoint derivation for all these available options. Due to the low-level development approach, these adjoint equations would then have to be implemented manually. This difficulty is compound by the fact that the model is under active development. That is, the derivation and implementation of the adjoint model would have to be updated with every change in the forward model. Fluidity also allows the user to embed Python (van Rossum et al., 2008) code to specify runtime functionality of the forward model. For example, initial and boundary conditions, source terms, and diagnostic functions can be specified entirely in Python. Since the mathematical equations of this functionality are not known a priori, the continuous adjoint can not be derived for these features beforehand.

The second alternative is to apply a low-level AD tool to Fluidity (see section 1.3.2). However, its source code has undergone years of development without consideration for the constraints that usually come with these AD tools. In particular the model is written in modern Fortran, and makes extensive use of advanced language features such as dynamic memory allocation, pointers, derived data types and function overloading. All of these are not supported by many AD tools. In addition linear equations are solved with the PETSc library (Balay et al., 1997, 2010), for which no differentiated version is available. For parallel execution, the model relies on MPI (Foster, 1995) whose differentiation is the topic of current research (Bücker et al.,

2004; Utke et al., 2009). All of these factors make applying currently available low-level AD tools to the whole model intractable.

Finally, the approach of deriving and implementing the discrete adjoint equations as described in section 1.3.3 can be applied. By lifting up the abstraction from the source code level to equation solves, the implementation details can be avoided, while parts of the automatic derivation of the adjoint system can be preserved. Section 2.7 presents the successful application of this approach to the shallow water model of Fluidity.

The second software package under consideration is FEniCS. In FEniCS, the user specifies the discrete problem in a high-level language, from which a dedicated finite element compiler generates efficient parallel code.

The main problem of applying the continuous adjoint approach to FEniCS is that the equations to be solved are provided as user input and hence not known a priori. Therefore, an automation of this approach must derive the continuous adjoint system from the user input. However, the current implementation of the high-level language of FEniCS has no abstraction for time discretisation and hence the time loop is implemented manually. A general reconstruction of the continuous equation from this time loop is not straightforward.

The application of a low-level AD tool is unfeasible for similar reasons to those for Fluidity: the control flow of a FEniCS model switches constantly between the driver program and the generated code. These difficulties are compound in the case of the Python interface of FEniCS where the code generation, compilation and execution happens dynamically at runtime. Furthermore, FEniCS is written in multiple different programming languages, exploits many modern programming language features and relies on external libraries, for example to solve the linear equations and to support parallelism, all of which make it difficult to apply a low-level differentiation tool.

The alternative of deriving the adjoint model from the discretised equations avoids the advanced code generation pipeline in FEniCS. This option is particularly suited for this case, because the discrete adjoint approach and the high-level language of FEniCS operate on the same abstraction level. Chapter 3 presents the application of libadjoint to FEniCS in detail, which results in an automatic and robust way to derive the adjoint models.

2.3. The fundamental abstraction

Motivated by the previous section and the advantages that the adjoint of the discretised equations offers (see section 1.3.3), the design of libadjoint is based on the fundamental abstraction that the forward model is a sequence of (linear or non-linear) equation solves.

For that, it is assumed that the forward model (1.2) can be written as a sequence of possibly non-linear equations that are solved consecutively. The solution u is a block vector in which each block corresponds to the solution of one equation. The exact choice of the blocks and what is considered as one equation is left free to the developer. In a time-dependent forward model, each block could for example contain the solution of a specific time level. In the application of libadjoint to the FEniCS system, presented in the following chapter, each block corresponds to the solution of a user specified variational problem.

While it would be possible to consider the forward model in the basic form (1.2), libadjoint expects the model cast in an extended, but equally expressive form:

$$A(u)u = b(u), \tag{2.1}$$

where A is a block matrix, b is a block vector and u is the solution block vector. For example, the basic form (1.2) can be cast into this form by writing:

$$Iu = Iu - F(u), \tag{2.2}$$

where I is the identity operator. The reason for using this extended form (2.1) is that it provides libadjoint explicit information about the linearity in equation (2.1). This can be an important optimisation as will be shown in section 2.5.3.1.

The fact that the equations are solved consecutively, reveals more information about the structure of equation (2.1). An equation solve can only depend on previously computed solutions, and hence A is lower-block-diagonal. For the same reason, the dependency arguments of a block row of A and b may only contain previously computed solutions and, for non-linear equations, the solution of the equation itself.

2.3.1. Examples

The following three examples demonstrate how a discrete PDE model can be cast into form (2.1).

2.3.1.1. Diffusion equation

The first example considers the steady diffusion equation given by:

$$-\nabla^2 u = f,$$

subject to appropriate boundary conditions. Discretising with the finite element method results in the linear system:

$$Du = Mf, \tag{2.3}$$

where $-D$ is the discretised diffusion operator, M is the mass matrix, and u and f are coefficient vectors for the solution and source term respectively.

Equation (2.3) can be trivially cast in the form of equation (2.1) by identifying $A = D$ and $b = Mf$.

2.3.1.2. Burgers' equation

In section 1.3.3 the Burgers' equation discretised with a forward Euler time-discretisation was written in a form which directly conforms to equation (2.1).

Now consider a backward Euler time-discretisation of the Burgers' equation (1.12). One obtains following assignments (compare to (1.16)):

$$\begin{aligned} u_0 &= g, \\ \frac{1}{\Delta t} M u_{n+1} - \frac{1}{\Delta t} M u_n + V(u_{n+1}) u_{n+1} + D u_{n+1} &= 0 \quad \text{for } n \in 0, \ldots, N, \end{aligned}$$

where the subscripts indicate the time levels, M is the mass matrix, $-D$ is the discretised diffusion operator, $V(u_{n+1})$ is the advection matrix assembled at the velocity u_{n+1}, and N indicates the number of time steps. Each time step consists of the solution of a non-linear equation, that can be solved for example with the Newton method. There are two choices how this system can be cast into form (2.1). Either each Newton iteration is

considered as one equation, which results in a set of linear equations (one equation per Newton iteration for each time step). Alternatively, the model can be directly considered as a sequence of non-linear equations. For that, define:

$$R(\cdot) \coloneqq \frac{1}{\Delta t} M + V(\cdot) + D.$$

The assignments (1.16) can then be cast into form (2.1) as:

$$\begin{pmatrix} I & & & \\ -\frac{1}{\Delta t}M & T(u_1) & & \\ & -\frac{1}{\Delta t}M & T(u_2) & \\ & & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} g \\ 0 \\ 0 \\ \vdots \end{pmatrix}.$$

2.3.1.3. Time-dependent diffusion equation with exponential source term

Now suppose the forward model approximately solves the time-dependent diffusion equation with an exponential source term:

$$\frac{\partial u}{\partial t} - \nabla^2 u = e^u,$$

subject to the initial condition $u(t = 0) = g$ and suitable boundary conditions. Discretising with the finite element method in space as above and the backward Euler method in time yields the iteration

$$\begin{aligned} u_0 &= g, \\ \frac{1}{\Delta t} M u_{n+1} - \frac{1}{\Delta t} M u_n + D u_{n+1} &= E(u_{n+1}) \quad \text{for } n \in 0, \dots, N, \end{aligned} \tag{2.4}$$

where subscripts denote time levels, Δt is the time step, M is the mass matrix, $-D$ is the discretised diffusion operator as before, E is the discretised exponential function, and N indicates the number of time steps.

The exponential operator can not be represented in the form $A(u)u$, and is therefore defined as part of the right hand side $b(u)$. Writing the itera-

tion (2.4) into form (2.1) yields:

$$\begin{pmatrix} I & & & & \\ -\frac{1}{\Delta t}M & \frac{1}{\Delta t}M + D & & & \\ & -\frac{1}{\Delta t}M & \frac{1}{\Delta t}M + D & & \\ & & \ddots & \ddots & \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} g \\ E(u_1) \\ E(u_2) \\ \vdots \end{pmatrix}. \tag{2.5}$$

Note that this representation is not unique: for example, the diffusion operator could be incorporated into the right hand side as part of $b(u)$. However, it will later be seen that the formulation (2.5) minimises the number of required operator differentiations as it unveils that E is the only non-linear operator.

In general, any discretised PDE may be cast in the form (2.1). For time-dependent simulations, u is generally a block-structured vector containing the unknowns for all time levels, A is a matrix with a lower-triangular block structure containing the discrete operators, and b is a block-structured vector containing the right-hand side terms of the equations.

However, this formulation does not imply that A and b are assembled and solved at once. For example, time-dependent forward models are typically solved from one time level to the next by iteratively assembling and solving one block-row of A and b. Similarly, libadjoint never demands to assemble the whole adjoint system. Instead, the adjoint equation is solved backwards from one time level to the next.

2.4. The discrete adjoint system

2.4.1. Derivation

The discrete adjoint model is derived from the forward model cast in form (2.1). For that, let $J(u)$ denote the real-valued functional of interest. Applying the definition of the adjoint equation (1.10) to the forward model (2.1) yields the general discrete adjoint system:

$$(A + G - R)^* \lambda = \frac{\partial J}{\partial u}^*, \tag{2.6}$$

where λ is the unknown adjoint solution, A is the left hand side of the forward model and G and R are defined as:

$$G \coloneqq \left(\frac{\partial A}{\partial u}\right)_{ijk} u_j, \qquad R \coloneqq \frac{\partial b}{\partial u}. \tag{2.7}$$

For clarification the definition of G is given in tensor index notation. The equivalent block-wise definition is:

$$G_{ik} = \sum_j \frac{\partial A_{ij}}{\partial u_k} u_j. \tag{2.8}$$

2.4.2. Examples

The general discrete adjoint system (2.6) can be used to derive the adjoint equations for any forward system cast in form (2.1). This is now demonstrated on two examples from section 2.3.1.

2.4.2.1. Adjoint of the diffusion equation

For the discretised diffusion equation (2.3), the operator A and the right hand side b do not depend on u, and hence $G = R = 0$. The resulting discrete adjoint system is therefore:

$$D^*\lambda = \frac{\partial J^*}{\partial u}.$$

2.4.2.2. Adjoint of the Burgers' equation

Next, consider the discretised Burgers' equation (1.18). For this example, the calculation of A^* is trivial and $R^* = 0$ as b does not depend on u. $(\partial J/\partial u)^*$ depends on the specific functional of interest. Therefore, attention is confined to the calculation of G. Once G is calculated, the calculation of G^* is trivial.

From the definition of G (2.7), the non-zero blocks of G are identified as:

$$G_{ik} \neq 0 \iff \text{block-row } i \text{ of } A \text{ depends on } u_k.$$

Analysing the left hand side of the Burgers' model (1.18) yields the block-sparsity pattern of the G matrix:

$$\begin{pmatrix} 0 & & & \\ * & 0 & & \\ & * & 0 & \\ & & \ddots & \ddots \end{pmatrix},$$

where $*$ denotes a non-zero block. To begin with, the top-left nonzero block, G_{21} is calculated. G_{21} records the dependency of the second row of A on variable u_1, i.e. the dependency of the equation that solves for u_1 on u_0. With the block-wise definition (2.8) the explicit form of G_{21} is obtained:

$$\begin{aligned} G_{21} &= \begin{pmatrix} \frac{\partial T(u_0)}{\partial u_0} & 0 & \dots \end{pmatrix} \cdot \begin{pmatrix} u_0 \\ u_1 \\ \vdots \end{pmatrix} \\ &= \frac{\partial T(u_0)}{\partial u_0} u_0. \end{aligned}$$

Both the mass operator M and diffusion operator D of $T(\cdot) = -\frac{1}{\Delta t}M + V(\cdot) + D$ do not depend on u_0. Therefore, these terms vanish in the differentiation, and thus G_{21} simplifies to:

$$G_{21} = \frac{\partial V(u_0)}{\partial u_0} u_0. \tag{2.9}$$

By repeating this derivation for the remaining components of G, the full G matrix is obtained:

$$G = \begin{pmatrix} 0 & & & \\ \frac{\partial V(u_0)}{\partial u_0} u_0 & 0 & & \\ & \frac{\partial V(u_1)}{\partial u_1} u_1 & 0 & \\ & & \ddots & \ddots \end{pmatrix}. \tag{2.10}$$

Combining the contributions gives the full left hand side operator of the discrete adjoint system (2.6):

$$(A+G-R)^* = \begin{pmatrix} I^* & \left(T(u_0) + \frac{\partial V(u_0)}{\partial u_0} u_0\right)^* & & \\ & \frac{1}{\Delta t} M^* & \left(T(u_1) + \frac{\partial V(u_1)}{\partial u_1} u_1\right)^* & \\ & & \frac{1}{\Delta t} M^* & \ddots \\ & & & \ddots \end{pmatrix}. \tag{2.11}$$

For these simple examples the derivation of the discrete adjoint model is not particular complicated. However, for a more complex forward model the manual derivation can become very extensive. In the case of the Burgers' equation, one might want to use a high-order time discretisation, or couple it to another PDE. Both change the dependency structure and introduce additional block-rows in the recasted forward model, which complicates the derivation of the discrete adjoint. Furthermore, this derivation needs to be repeated every time the structure of the forward model changes, for example if the forward model is configurable at runtime.

Despite its complication, the key property of this adjoint derivation is that it follows a completely prescriptive process: once the symbolic description of the discrete forward system in form (2.1) is available, the description of the associated adjoint system follows. Furthermore, if all block operators that occur in this symbolic description can be assembled, then each adjoint equation can be assembled and solved. In the Burgers' example, the assembly of the adjoint operator (2.11) requires the block operators $I^*, M^*, T^*(\cdot)$ and $(\frac{\partial V(\cdot)}{\partial \cdot}\cdot)^*$ and the adjoint right hand side requires the partial derivatives of J.

The library libadjoint automates exactly that symbolic derivation and facilitates the assembly and the computation of the adjoint equations. The application of libadjoint to a forward model can be divided into three steps:

Annotation The annotation builds an internal tape with high-level information about each equation solve that was performed during the forward model. Once the annotation is finished, libadjoint has a symbolic representation of the forward model in the form of (2.1). From this it can automatically derive a symbolic description of the associated adjoint system.

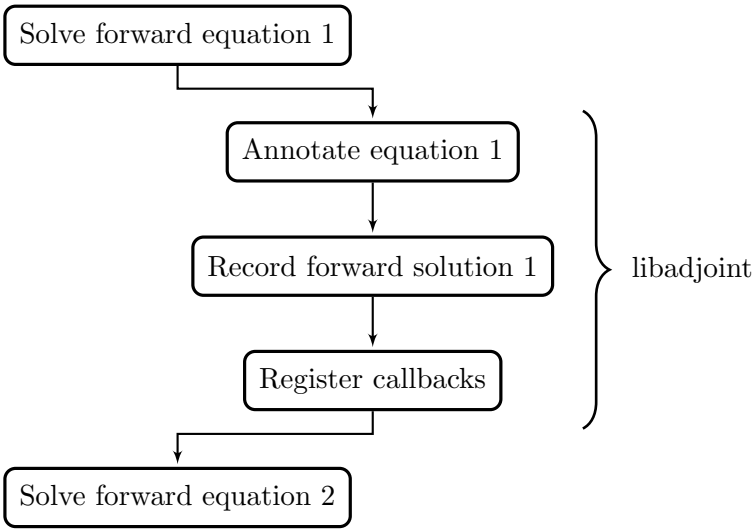


Figure 2.1.: A typical implementation of the three required steps for applying libadjoint to a forward model.

Recording the forward solutions The adjoint system depends in general on the forward solution. Consequently, libadjoint offers the functionality to record forward solutions that are required for assembling the adjoint operators.

Callback registration In order to solve the adjoint system, libadjoint must be able to assemble the block operators that occur in the symbolic representation. This is achieved by attaching function callbacks to the symbolic operators. The model developer must implement these operators according to libadjoint's interface. Similarly, libadjoint uses function callbacks to perform algebra operations, such as vector addition or matrix multiplication.

Once the necessary information for these three steps has been provided, libadjoint can automatically derive, assemble and solve the adjoint system. The order in which the above steps are executed is free to the model developer. A common choice is to perform all three steps as the equation solve happens, see figure 2.1. These three steps will now be explained in further detail.

2.5. Using libadjoint

This section discusses the necessary steps to apply libadjoint to a forward model. libadjoint is open-source and can be downloaded at `https://launchpad.net/libadjoint`. The core of libadjoint is implemented in the C programming language for portability reasons, however additional interfaces to Fortran 90 and Python are available.

The code examples in the following sections are presented in Python. Similar implementations in C and Fortran 90 can be found in the test directory of the libadjoint code base. A detailed description of all available libadjoint functions can be found in the manual (Farrell and Funke, 2011).

2.5.1. Annotation

The purpose of the annotation is to build a tape with high-level information about each equation solved in the forward model. For that, libadjoint offers library functions that enable the model developer to express:

- the block structure of an equation;
- any non-linear dependencies of each block.

Each equation of the forward system is expressed using these functions and passed to libadjoint.

The following code example shows an annotation of the first block equation of the discretised Burgers' equation (1.18), with libadjoint specific functions marked red.

```
# Initialisation
adjointer = Adjointer()

# Create a symbolic representation of the identity operator
I = Block("IdentityOperator")
# Create a symbolic representation of the
# velocity vector at timestep 0
u = Variable('u', timestep = 0)
# Create a symbolic representation of the equation Iu = 0
e = Equation(u, [I], [u])

```

```
# Add the equation to the annotation
adjointer.register_equation(e)
```

Note, that there is no need to annotate the right hand side of the Burgers' forward model (1.18): it does not have any dependencies on u and is hence irrelevant for the adjoint system (however, it is required for the replay functionality that will be discussed in section 2.5.6). The annotation code for the remaining equations of the Burgers' model (1.18) is similar and simply mimics the structure of the discrete system.

In a practical implementation, the annotation calls are typically placed just before or after the associated equation is solved, see figure 2.1. The reason is that at this point usually all required dependency information is available. This also has the advantage that the annotation only happens if the associated equation is actually solved, which ensures that the annotation is always consistent with the forward model. This is particularly important for forward models that solve different equations depending on user configuration.

Once the annotation has been executed, libadjoint has an internal symbolic representation of all forward equations and their dependencies in form (2.1).

2.5.2. Recording the forward solutions

libadjoint needs to have access to the forward solutions in order to assemble and solve the adjoint equations. For that, the library offers functions to record the values of the forward variables. These function calls are naturally best placed directly after the equation solve in the forward model, see figure 2.1

Continuing the example code above, the following code shows how the initial velocity value in the Burgers' example is recorded:

```
# Record the forward solution for the adjoint assembly
# Here "solution" contains the result of the associated
# forward equation
adjointer.record_variable(u, MemoryStorage(solution))
```

Storing the forward solutions for large models can require a significant amount of memory and so it is important to keep the memory footprint as low as possible. For that, libadjoint applies two strategies. Firstly,

it only demands forward solutions on which the adjoint system depends and so only these need to be recorded. Secondly, the library automatically deletes forward solutions as soon as they are not required for any remaining calculations during the adjoint computation.

In addition, libadjoint can use checkpoints that allow the recomputation of forward solutions instead of their storage. These checkpointing strategies will be discussed in section 2.6.2.

2.5.3. Function callbacks

2.5.3.1. Operator callbacks

With the annotation, libadjoint can derive the structure of the adjoint system. It cannot, however, assemble and solve the adjoint equations. The operators mentioned in the annotation are at this point just abstract symbols to be manipulated. This problem is resolved by associating functions to each of the operators that perform the required tasks. The model developer must implement these functions with a defined interface, and attach them as callbacks to the symbolic operators.

The specific required functionality that needs to be implemented depends on the linearity of the operator and where it occurs in the annotation. From the general discrete adjoint system (2.6) it can be seen that libadjoint needs callbacks to assemble each the terms A^*, G^*, R^* and $(\partial J/\partial u)^*$. Firstly, consider the term A^*. Its assembly requires callbacks for each operator X that occurs in the annotation matrix A to:

- assemble the operator X^* or a matrix free representation thereof, if the operator X occurs on the diagonal of A;
- compute its action on a given block vector v, i.e. X^*v, or a matrix free representation thereof, otherwise.

Usually, the implementation of these callbacks is straightforward, since the non-transposed operators are already available in the forward model. Their Hermitian transpose is either obtained manually or by using the built-in functionality of many linear algebra packages. Alternatively, the Hermitian transpose action callbacks can be derived by algorithmic differentiation as described in section 2.5.5. If the operators are symbolic objects, as is the

case of the FEniCS framework presented in the next chapter, the required callback operations are obtained by simple symbolic manipulations.

Next, consider the assembly of the G^* contributions. For a non-linear operator $X(v)$ that occurs as part of the annotation matrix A, callbacks are required to

- compute the Hermitian transpose of the Jacobian contracted with a given block vector c, i.e. $((\partial X/\partial v)\, c)^*$, or a matrix free representation thereof, and
- compute its action on a given block vector w, i.e. $((\partial X/\partial v)\, c)^*\, w$, or a matrix free representation thereof.

The R^* term assembly requires callbacks for any non-linear operator $X(v)$ that is part of the right hand side b in the annotation system (2.1) to

- compute the Hermitian transpose of the Jacobian, $(\partial X(v)/\partial v)^*$, or a matrix free representation thereof.

In practice, some of these derivatives are usually implemented in the forward model, for example if the Newton method is used to solve the non-linear equations. Otherwise, section 2.5.5 shows how the action callbacks can be obtained with automatic differentiation. If the operators are represented symbolically, as is the case in the following chapter, these callbacks are again obtained by simple symbolic manipulations.

Finally, the right hand side of the adjoint system (2.6) requires a callback to

- compute the functional derivative $(\partial J/\partial v)^*$ for a given block vector v, or a matrix free representation thereof.

If this function is not available in the forward model, it can either be derived and implemented manually, or obtained with the help of automatic differentiation.

The knowledge about linear and non-linear operators allows libadjoint to minimise the amount of code for which the derivative callbacks are required. For example, the highlighted operations in the following Burgers' iteration from section 1.3.3 mark the code that needs differentiating if an AD tool is

applied to the whole model:

$$\begin{gathered} u_0 = g, \\ \frac{1}{\Delta t} M u_{n+1} - \left(\frac{1}{\Delta t} M - V\left(u_n\right) - D \right) u_n = 0 \quad \text{for } n \in 0, \dots, N, \\ J(u_1, u_2, \dots). \end{gathered}$$

With libadjoint's annotation, the required derivatives reduce to only the non-linear operators, i.e.:

$$\begin{gathered} u_0 = g, \\ \frac{1}{\Delta t} M u_{n+1} - \left(\frac{1}{\Delta t} M - \boxed{V\left(u_n\right)} - D \right) u_n = 0 \quad \text{for } n \in 0, \dots, N, \\ \boxed{J(u_1, u_2, \dots)}. \end{gathered}$$

The following code example implements and registers the identity callback for the Burgers' equation (1.18). Since the identity operator appears on the diagonal, libadjoint requires a callback that assembles the corresponding block or a matrix free representation of it. An implementation that uses the numpy data structures (Oliphant, 2006) for vectors and matrices is:

```
# Define a callback following libadjoints specification
def identity(dependencies, values, hermitian,
             coefficient, context):
    # == "dependencies" and "values" ==
    # Lists of forward solutions that are required for
    # the assembly of nonlinear operators. In this case empty.
    # == "hermitian" ==
    # A boolean which indicates of the original or the
    # tranpose operator is to be returned.
    # == "coefficient" ==
    # a scaling coefficient. libadjoint sometimes adds or
    # subtracts operators in which case "coefficent" is != 1.
    # == "context" ==
    # User defined context information that can be passed
    # to the callback

    # Create the identity matrix with 100 entries
    id_mat = Matrix(coefficient*numpy.eye(100))
    # libadjoint also needs a vector to store the
    # solution of the equation
    id_vec = Vector(numpy.zeros(100))
    return (id_mat, id_vec)

# Attach the callback to the identity block that was defined
# in the annotation code in section 2.5.1.
I.assemble = identity
```

Of course, in practice one would usually implement a matrix free representation of the identity operator instead. The callback function parameters contain all required information to perform the assembly. In particular, for non-linear operators, such as the advection operator V in the Burgers' model (1.18), libadjoint passes the non-linear dependencies to the callback. In addition, the model developer can attach arbitrary data to the symbolic operators which is then passed to the associated callbacks via the `context` argument. In the example above, it might be useful to store the dimension

of the matrix in the context. Other examples are the time step number, the current time, or accumulative statistics about the model. Further details about the callback interface can be found in the libadjoint manual (Farrell and Funke, 2011).

2.5.3.2. Callbacks for algebraic operations

libadjoint needs to perform certain algebraic operations on vectors and matrices, such as addition, scaling and matrix-vector multiplication. For solving the equations, the library also needs to have access to appropriate solver routines.

Similar to the operator callbacks from the previous section, libadjoint relies on user defined callback functions for these algebraic operations. The set of operations that the adjoint model developer must implement depends on the functionality used. The bare necessities include callbacks for matrix-vector multiplication, and creating, deleting and adding vectors and matrices, as they are needed to assemble the adjoint equations. The automatic computation of the adjoint solution, presented in the next section, is based on a callback that implements an equation solver. For the checkpointing functionality discussed in section 2.5.6, libadjoint relies on additional callbacks for creating checkpoints and to read and write forward solutions from and to disk. Finally, the debugging tools discussed in section 2.5.6 require further callbacks such as the computation of the vector norm and to fill a vector with random values.

While the implementation of these callbacks require a certain initial overhead of applying libadjoint to a new forward model, it allows libadjoint to remain lightweight and flexible towards any specific data structures. For data types for common libraries such as PETSc (Balay et al., 1997, 2010) and numpy (Oliphant, 2006), libadjoint comes with built-in implementations of these callbacks, which can be directly used by the adjoint model developer.

2.5.4. Solving the adjoint system

After the steps described above, libadjoint has the necessary information about the forward model to derive, assemble and solve the associated adjoint system. The following code example demonstrates the user interface. The

user first defines the functional of interest for which the adjoint system is to be derived. Then, the adjoint solutions can be computed by looping backwards over the equation indices, solving the associated adjoint equation and recording the result:

```
# Define a class that implements the functional of interest.
class MyFunctional(Functional):
    def derivative(self, adjointer, variable, dependencies,
                   values):
        # This function must return the partial derivative of
        # the functional with respect the forward variable
        # "variable". It is used by libadjoint to add the
        # adjoint source term to the right hand side of the
        # adjoint equations.
        ...

J = MyFunctional()

# Loop backwards over all adjoint equations
for i in range(adjointer.equation_count)[::-1]:
     # Derive, assemble and solve the i'th adjoint equation
     (var, val) = adjointer.get_adjoint_solution(i, J)
     # Record the adjoint solution
     adjointer.record_variable(var, MemoryStorage(val))
```

The core function `adj_get_adjoint_solution` performs four main steps. First, it derives the structure of the adjoint equation using the annotation. Second, if the equation contains any non-linear blocks it loads the required dependency values from the recorded forward solutions. It then calls the associated assembly and action callbacks to assemble the adjoint equation, and finally passes the result to the solver callback and returns the solution.

2.5.5. Callback generation with algorithmic differentiation

This section demonstrates how the required action operator callbacks (section 2.5.3.1) can be automatically generated from C and Fortran code by applying a low-level AD tool to the associated assembly routines of the forward model. This is unnecessary in the case where the callbacks manage

symbolic representations of the operators, such as in the FEniCS framework described in the next chapter.

All of the action operator callbacks listed in section 2.5.3.1 can be generated using algorithmic differentiation.

For example, assume the required callback is the Hermitian transpose action X^*v for a operator X that is part of the annotation matrix A, and a block vector v. For that, one implements a functional computing $f(v, c) := v^*(Xc)$ where c is a block vector. Applying an AD tool in reverse mode to compute the derivative of the functional f with respect to c yields the sought function $\mathrm{d}f/\mathrm{d}c = X^*v$.

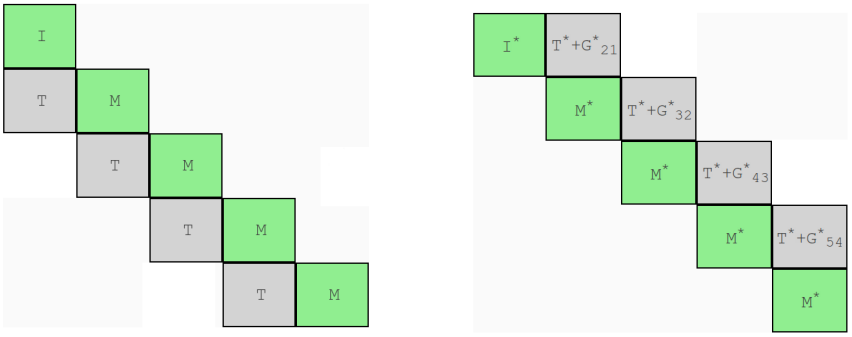
A more complex computation arises for a non-linear operator $X(v)$ that is part of the annotation matrix A. For this, an action callback is required that computes $((\partial X/\partial v)c)^*w$ for block vectors v, w and c. This is achieved by first writing a functional $f(v, w, c) \coloneqq (X(v)c)^*w$. Applying and AD tool in reverse mode to compute its derivative with respect to v yields the required callback function. The remaining action callbacks for the contributions of R^* and the right hand side $(\partial J/\partial u)^*$ are obtained in a similar way.

While these assembly routines are typically amenable to AD tools, the restrictions of the low-level AD tool remains. In particular, the application of these tools becomes more complicated, if the assembly routines depend on external libraries such as MPI for parallelisation.

2.5.6. Verification and debugging

Special attention has been drawn to provide verification and debugging tools for every stage of the adjoint model development with libadjoint. As a result, five features have been implemented that allow the quick identification and location of problems during the development and production stage of the adjoint model.

Firstly, libadjoint offers to save detailed information about its current state to an HTML file, that can be viewed in any web browser. This file contains various information including a visualisation of the annotated forward and its derived adjoint model. An example HTML output produced from the Burgers' annotation is shown in figure 2.2. In addition, the HTML output shows the registered callbacks and the forward and adjoint variables that are currently recorded. If a checkpointing scheme is used (see



(a) The forward matrix A (compare to equation (1.18)).

(b) The adjoint matrix $(A + G - R)^*$ (compare to equation (2.11)).

Figure 2.2.: The HTML output of libadjoint on the Burgers' equation annotation. The green background denotes the diagonal blocks; and the grey background marks off-diagonal, nonzero blocks.

section (2.6.2)), further information about the checkpointed equations and their associated variables is included. This debugging tool is useful throughout the adjoint model development. In particular, it greatly facilitates the annotation process, since it allows a direct comparison of the annotation and the discrete forward model in form (2.1).

Secondly, the library can verify the consistency of the annotation with the forward model. This test is based on the idea that libadjoint can use the annotation not only to derive and solve the adjoint system, but also to rerun the forward model. While doing so, it is asserted that the computed solutions are equivalent to the solutions that have been previously recorded from the forward model. For this feature to work libadjoint needs to have additional operator callbacks implemented that assemble the non-Hermitian transpose operators mentioned in the annotation and the right hand side of the forward equation (2.1). A successful execution of this test is a strong indicator that both the annotation and the implementation of the non-Hermitian transpose operators are correct.

The third debugging feature is used to verify the Hermitian transpose version of the callbacks. Given an operator X and its adjoint implementation X^*, libadjoint generates two random block vectors v, w with appropriate

dimensions and checks whether following condition holds:

$$\langle v, Xw \rangle = \langle X^* v, w \rangle . \tag{2.12}$$

An incorrect implementation of the Hermitian transpose is typically detected by repeating this test with a few different random vectors.

Fourthly, libadjoint includes a test to check the implementation of callbacks that involve derivatives. The test applies a first-order Taylor expansion and asserts that the convergence order is as expected. The Taylor remainder convergence test is based on the observation that, given a non-linear operator $X(v)$, a dependency vector v and an arbitrary perturbation δv to the initial conditions p:

$$\|X(v + h\delta v) - X(v)\| = \mathcal{O}(|h|) \quad \text{as } h \to 0,$$

but that:

$$\left\| X(v + h\delta v) - X(v) - h\frac{\partial X(v)}{\partial v}\delta v \right\| = \mathcal{O}(|h|^2) \quad \text{as } h \to 0.$$

For a given vector v, libadjoint generates a pseudorandom perturbation vector δv with each component uniformly distributed and verifies the convergence orders for $h \to 0$. This test is extremely sensitive to even slight errors in the implementation of the derivative callbacks. A typical case where this test fails is if a non-linear derivative callback is generated with an AD tool, but has not been reapplied after changing the non-linear operator.

Finally, the correctness of the adjoint solution can be verified by applying the Taylor remainder convergence test to the reduced functional of interest $\tilde{J}(m)$ (definition (1.4)), i.e. the functional of interest considered as a pure function of a parameter m. After computing the functional gradient $\nabla\tilde{J}(m)$ from the adjoint solution with equation (1.11), the Taylor expansion states that:

$$\left| \tilde{J}(m + h\delta m) - \tilde{J}(m) \right| = \mathcal{O}(|h|) \quad \text{as } h \to 0, \tag{2.13}$$

but:

$$\left| \tilde{J}(m + h\delta m) - \tilde{J}(m) - h\delta m^* \nabla\tilde{J}(m) \right| = \mathcal{O}(|h|^2) \quad \text{as } h \to 0. \tag{2.14}$$

```
Traceback (most recent call last):  [...]
  libadjoint.exceptions.LibadjointErrorNeedCallback:
  Could not find callback ADJ_BLOCK_ASSEMBLY_CB for operator M.
```

(a) libadjoint throws this exception if a callback routine is missing (here the assembly callback for the mass matrix from the Burgers' example) for one of the operators in the annotation.

```
Traceback (most recent call last):  [...]
  libadjoint.exceptions.LibadjointErrorNeedValue:
  Need a value for u:1:0:Forward, but don't have one recorded.
```

(b) This error occurs during the adjoint computation if an adjoint equation depends on a forward variable (here the forward solution of the second timestep and first non-linear iteration from the Burger's example) which has not been recorded.

```
Comparing u:1:0:Forward against previously recorded value:
norm of the difference is 1.648e+00 (> tolerance of 0.000e+00).
```

(c) During libadjoint's replay run, this error message indicates an inconsistency between original forward model and the annotated model.

Figure 2.3.: Python exceptions thrown by libadjoint for typical errors during the adjoint model development.

The second-order convergence is very sensitive to implementation errors in the adjoint, and rigorously checks that the computed functional gradient is consistent with the discrete forward model.

With the aid of these debugging and verification tools the development of the adjoint model becomes a step wise process after each of which its correctness can be verified.

2.5.7. Error handling

One of the design concepts of libadjoint is to always check the prerequisites before performing a task. In case one of the requirements is not fulfilled, libadjoint stops its execution and informs the user. Some examples of errors that typically occur during the development phase are listed in figure 2.3.

In particular, libadjoint returns an error if:

- a forward/adjoint solve contains an operator for which the required callback is not registered;
- an adjoint solve depends on a forward variable which was not recorded;

- the replay test finds an inconsistency between the original forward solution and the recomputation with libadjoint;

- one of the Taylor remainder convergence tests from the previous section exceeds the defined tolerance.

A detailed list of the errors and warnings can be found in the libadjoint manual (Farrell and Funke, 2011).

In C and Fortran 90, the error handling is implemented by returning defined error codes; in Python, libadjoint throws an exception. This error handling further greatly simplifies the development process of the adjoint model: in many cases the developer can apply a "try and error" approach since the error messages given by libadjoint are often expressive enough to indicate what needs to be implemented next.

2.6. Description of the core algorithms

This section discusses some details about the core algorithms used in libadjoint.

2.6.1. Deriving the discrete adjoint equations

This section describes the core of libadjoint that derives and assembles the discrete adjoint system. The assembly of the adjoint system happens equation by equation with backward substitution, just as the forward system is solved equation by equation using forward substitution. That is, the k'th adjoint equation, the equation that solves for the adjoint solution λ_k, can be written as (compare to equation (2.6)):

$$(A + G - R)^*_{kk}\lambda_k = \frac{\partial J}{\partial u_k}^* - \sum_{l,l>k} (A + G - R)^*_{kl}\lambda_l. \tag{2.15}$$

Algorithms 1 and 2 describe the procedures which libadjoint follows to assemble the left and right hand side of this equation, respectively. In order to avoid unnecessary assembly calls, these algorithms check in advance if an operator evaluates to zero, which occurs if the derivative of an operator is taken with respect to a variable on which it does not depend.

ALGORITHM 1: Assembly of the left-hand side of the equation for λ_k

Data: the index k of the adjoint equation to assemble
Result: the left-hand side of the k'th adjoint system `lhs`
$G^*_{kk} \leftarrow 0$
$R^*_{kk} \leftarrow 0$
if *block row k of A has a dependency on* u_k **then**
  for *all blocks* A_{kj} *that depend on* u_k **do**
    $G^*_{kk} \leftarrow G^*_{kk} + \left(\frac{\partial A_{kj}}{\partial u_k} u_j\right)^*$
  end
end
if b_k *has a dependency on* u_k **then**
  $R^*_{kk} \leftarrow \left(\frac{\partial b_k}{\partial u_k}\right)^*$
end
`lhs` $\leftarrow A^*_{kk} + G^*_{kk} - R^*_{kk}$

2.6.2. Optimal checkpointing

The adjoint operator in equation (2.6) consists of the term $(A + G - R)^*$. For a non-linear forward model, A will have dependencies on the forward solution and hence A^* as well. Similarly, G^* and R^* may have complex dependencies on the forward solution. As a consequence, different parts of the forward solution must be available at different times when assembling the adjoint equations. While the straightforward approach of storing the whole forward solution is unproblematic for steady state problems, it can become prohibitively expensive for time-dependent simulations, since the storage requirement increases proportionally to the number of time steps. For example, storing a forward solution with 10^7 unknowns for 10^5 time steps on a standard computer requires over 7 TB. It is therefore apparent that the available storage would greatly constrain the number of time steps for which a real-world adjoint simulation could be run.

One way to circumvent this problem is to use temporal interpolation. Here, instead of storing all forward solutions, only solutions at selected time levels are kept. When the adjoint assembly requires an intermediate forward solution, a temporal interpolation scheme is used to approximate its value. Although the generality of this approach is questioned in Gunzburger (2003), it is sometimes used in hand-written adjoint codes. The adjoint of the NEMO ocean model is one such example (Vidard et al., 2008). Their experience shows that the necessary code manipulation "requires deep

ALGORITHM 2: Assembly of the right-hand side of the linear system for λ_k

Data: the index k of the adjoint equation to assemble
Result: the right-hand side of the k'th adjoint system `rhs`
`rhs` $\leftarrow \frac{\partial J}{\partial u_k}^*$
for *all blocks* $A_{lk} \neq 0$, $l \neq k$ **do**
  `rhs` $\leftarrow$ `rhs` $-A_{lk}^* \lambda_l$
end
for *all equations* l *that depend on* u_k, $l \neq k$ **do**
  for *all blocks* A_{lj} *that depend on* u_k **do**
    `rhs` $\leftarrow$ `rhs` $- \left(\frac{\partial A_{lj}}{\partial u_k} u_j \right)^* \lambda_l$
  end
  for *all* l *such that* b_l *depends on* u_k, $l \neq k$ **do**
    `rhs` $\leftarrow$ `rhs` $+ \frac{\partial b_l}{u_k}^* \lambda_l$
  end
end

knowledge of the original program and of the underlying equations ... [and] introduces approximation errors into the computed derivatives, whose mathematical behavior is unclear" (Tber et al., 2007, §4.3).

The problems associated with interpolation may be circumvented using checkpoints, from which the forward simulation can be restarted to recompute missing forward solutions. In the simplest checkpointing strategy, the complete time interval is split into a small number of equidistant subintervals and a checkpoint is stored at the beginning of each interval. When an intermediate forward solution is required for the adjoint assembly, the forward simulation is restarted from the closest preceding checkpoint and run until the required value is recomputed. An extension known as multi-level checkpointing applies this idea recursively: the time interval of the partial forward integration is again split into several subintervals on which checkpoints are stored, and so on. This multi-level checkpointing is successfully used in the MITgcm ocean model (Heimbach et al., 2005).

Griewank (1992) proposed a strategy related to the multi-level approach in which the checkpoint distribution is based on a binomial interval splitting. By reusing the available checkpoint slots, a logarithmic growth of temporal and computational complexity is achieved, which is proven to be optimal (Grimm et al., 1996). Walther and Griewank (2004) showed that this approach is indeed advantageous over the multi-level checkpointing strategy

and published a library named *revolve* that facilitates its implementation (Griewank and Walther, 2000).

Both multi-level checkpointing and *revolve* allow the adjoint model user to balance the storage and computational complexity to their needs. However, this flexibility comes with additional development effort: the control flow constantly switches context between the forward and adjoint main loops, making the implementation of checkpointing more difficult than the temporal interpolation.

While this can involve extensive development effort for a hand-written adjoint, the abstraction of AD is powerful enough to automatically generate the adjoint control flow according to the checkpointing strategy. Since the tape can be used to restart the simulation from a checkpoint, the checkpointing logic can be implemented within the AD tool. However, because the abstraction of low-level AD considers the model as a sequence of elementary instructions, it does not natively know about the concept of timestepping, which is required for efficient checkpointing. This information must therefore be provided by the model developer, often by explicitly adding AD-specific directives to the forward code. AD tools that offer checkpointing include TAF (Giering and Kaminski, 2002) and ADOL-C (Kowarz and Walther, 2006).

As with low-level AD, libadjoint is sufficiently powerful to execute the required operations for checkpointing entirely within the library. In particular, the annotation and the supplied callbacks allow libadjoint to restart the simulation from a checkpoint. But in contrast to low-level AD tools, libadjoint's annotation also provides the required time level information. This makes checkpointing with libadjoint available for almost no extra model development effort: the only requirement is to implement the callbacks for creating and deleting checkpoints.

An implementation of this optimal checkpointing scheme within libadjoint faces several challenges. Firstly, it must be decided at which time levels the checkpoints should be placed. libadjoint uses the optimal checkpointing algorithm proposed by Griewank (1992) by interfacing to the *revolve* library (Griewank and Walther, 2000). The following section introduces this scheme.

Secondly, it must be determined which forward solutions must be contained in a checkpoint. The checkpoints are used to restart the forward

model, but they also need to store certain forward solutions for the adjoint computation. This problem will be discussed in detail in section 2.6.2.2.

2.6.2.1. Binomial checkpointing

This section briefly describes the binomial checkpointing strategy implemented in *revolve*. It manages the control flow between forward and adjoint solves and finds the optimal checkpoint positions to minimise recomputation costs. The basic strategy, which assumes that the number of time levels is known a priori, is shown in algorithm 3. Clearly, the storage requirement of this algorithm is limited by the data required to store c checkpoints. The exact storage size depends on which forward solutions need to be included in the checkpoints. An execution of this algorithm for $N = 10$ and $c = 3$ is visualised in figure 2.4.

The computational expense of algorithm 3 depends on the number of recalculations in the **advance** step. Therefore, the question arises where to optimally locate the checkpoints in order to minimise the number of recomputations. To answer this question, let t be the maximum number of how often any forward equation is solved in the **advance** steps. This number yields an upper bound for the computational cost. For example, if $t = 2$ the computational cost of the **advance** steps is less or equal than two full forward simulations. Now the question can be rephrased to: how large may the number of time levels N be for a fixed t? Griewank (1992) showed that

$$N \leq \beta(c,t) \coloneqq \binom{c+t}{c}. \tag{2.16}$$

ALGORITHM 3: Binomial checkpointing

Data: number of time levels N; number of available checkpoints c
Result: the adjoint solution
Checkpoint equation 1
for $k = N, 1, -1$ **do**
  advance Run the forward model from the nearest checkpoint until equation $k-1$. On the way, fill all free checkpoints using the binomial formula (2.16) as described below.
  reverse Solve the k'th forward and adjoint equation. If equation $k-1$ is a checkpoint, free it.
end

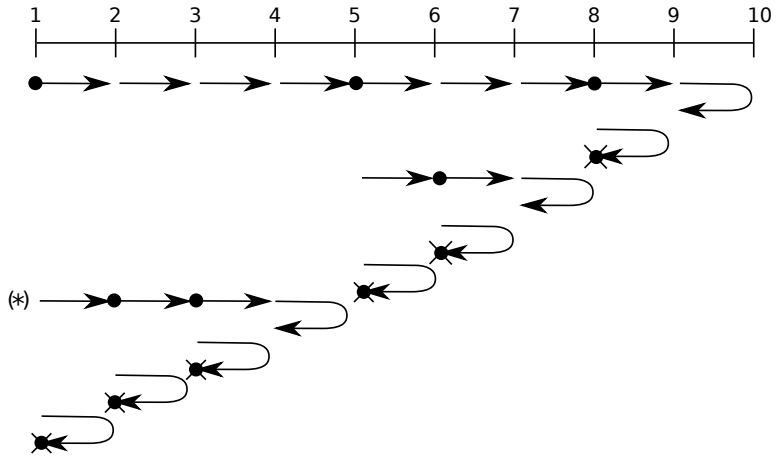


Figure 2.4.: Visualisation of binomial checkpointing strategy applied to $N = 10$ time levels and $c = 3$ checkpoints. The arrows show the temporal flow of the calculations: an arrow to the right (left) indicates the solution of one time step of the forward (adjoint) model. A (crossed) dot indicates the creation (deletion) of a checkpoint.

This result is the key to find the optimal placement of checkpoints. Initially equation (2.16) is used to determine the smallest t that satisfies $N \leq \beta(c, t)$ for the given number of checkpoints c and time levels N. For the example in figure 2.4, $t = 2$ satisfies this condition. To find the position for the second checkpoint (the first checkpoint is always for the first equation) in the example in figure 2.4, it is observed that the recomputations of the first checkpoint slot (marked with $(*)$ in figure 2.4) must be computable with at most $t - 1$ calculations per equation. The maximum number of time levels that can be computed with that restriction is $\beta(c, t-1) = \beta(3, 1) = 4$, and so the second checkpoint is placed at equation 5. The position of the third checkpoint is determined in the same way, but needs to take into account that one checkpoint less is available. Therefore, the third checkpoint is placed $\beta(c-1, t-1) = \beta(2, 1) = 3$ time levels later. By recursively applying this algorithm the full checkpoint positioning in figure 2.4 can be reconstructed.

This checkpointing distribution makes it possible to run large, time-dependent adjoint models because β increases rapidly with larger c or t values in equation (2.16). For example, a simulation with $30,000$ time levels and 55 checkpoints results in a total cost of the **advance** steps that is at

most three times the cost of one full forward run. With four times the cost of one forward model over $455,000$ time steps can be performed. Furthermore Griewank (1992) showed that under certain assumptions a trade-off between spatial and computational complexity can be obtained such that both the spatial and computational cost increase logarithmically with the number of time levels. The optimality of the method has been proven by Grimm et al. (1996).

The *revolve* library implements two extensions of the basic binomial checkpointing strategy, both of which are supported by libadjoint. The first extensions allows the user to store some checkpoints in memory and some on disk. Saving checkpoints in memory has the benefit of a fast access time but its capacity is often limited. Disks are slower but usually have much larger storage capacity. The extension developed by Stumm and Walther (2009) manages the storage destination for the checkpoints. The second extension removes the assumption that the number of time levels is known a priori in algorithm 3. This is particularly important for forward models that apply time adaptive schemes where the number of time levels is only known after the first time integration. The checkpoint distribution for this case has been investigated by Stumm and Walther (2010).

2.6.2.2. Determination of required forward solutions in a checkpoint

The checkpoints in algorithm 3 are created for two purposes, each of which can introduce dependencies on different forward solutions, that the checkpoint must contain. Firstly, they are used in the **advance** step to restart the forward model from their associated time level. Secondly, the **reverse** step uses them to access dependencies of the adjoint equation on forward variables that are not recomputed by the **advance** step.

This makes it difficult to determine which forward variables need to be included in a checkpoint. However, libadjoint's annotation is expressive enough to derived these dependencies automatically. Consequently, once the annotation of the forward model is complete, libadjoint handles the entire process of filling and freeing checkpoints internally, without user intervention.

However, usually the annotation happens simultaneously with the forward solve (see figure 2.1). Therefore, during the annotation libadjoint can not determine which forward solutions to include in a checkpoint, since a future forward equation might depend on any previously computed solution. In that case, libadjoint requests the adjoint model developer to provide a list of forward solutions that need to be included in a checkpoint. The following checkpoint definition specifies which solutions libadjoint expects in this list.

Definition 2. *Let c be the index of a forward equation which is to be checkpointed. A checkpoint is a set of forward solutions u_i with $i < c$, each of which is required for either:*

1. *solving a forward equation $\geq c$ or*
2. *evaluating an adjoint right hand side $\frac{\partial J}{\partial u_k}^*$ with $k \geq c$.*

Note that the first purpose of a checkpoint, being able to restart the forward model, is trivially fulfilled by the first condition in definition 2. However, the definition does not contain any explicit requirements to store dependencies of the corresponding adjoint equations, despite its right hand side. The following theorem shows that these dependencies are implicitly contained in the checkpoints.

Theorem 1. *A checkpoint for equation c contains all forward solutions u_i, $i < c$ that are required to assemble any adjoint equation $\geq c$.*

Proof. Without loss of generality, the forward model is considered in the form:

$$F(u) = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \end{pmatrix} (u) = 0,$$

where each F_l corresponds to one equation in the forward model with solution u_l. The forward propagation of information in the forward model ensures that any F_l only depends on forward solutions u_i with $i \leq l$.

Now, let $k \geq c$. The k'th adjoint equation, the adjoint equation that solves for λ_k, is (compare to equation (2.15)):

$$\sum_{l,l \geq k} \left(\frac{\partial F}{\partial u} \right)^*_{kl} \lambda_l = \frac{\partial J}{\partial u_k}^* . \tag{2.17}$$

The proof is complete if the checkpoint c contains all forward solution dependencies u_i, $i < c$ for this equations.

The required dependencies to assemble the right hand sides are trivially contained in the checkpoint because of the second requirement in definition 2.

Now consider the dependencies of the left hand side in equation (2.17). Assume that there is a $(\partial F/\partial u)_{lk}$, $l \geq k$ that depends on u_i with $i < c$. That is, the l'th forward equation $F_l = 0$, with $l \geq k \geq c$ depends on forward solution u_i. The first requirement in definition 2 ensures that u_i is contained in the checkpoint. □

This theorem greatly simplifies the task of providing the list of forward solutions that must be included in a checkpoint since the adjoint operator dependencies can be ignored.

2.7. Examples

The following examples demonstrate the application of libadjoint to two models that are included in Fluidity. Another application of libadjoint can be found in the following chapter, where the library is applied to the FEniCS system.

2.7.1. Burgers' model

As part of its developer training documentation, Fluidity includes a small example solver for the Burgers' equation (1.12). The performed solutions steps are similar to the iteration (1.16), but uses two Picard iteration per time step to deal with the non-linear advective term:

$$
\begin{aligned}
u_0 &= g, \\
\frac{1}{\Delta t} M u^0_{n+1} &= \left(\frac{1}{\Delta t} M - V(u_n) - D \right) u_n && \text{for } n \in 0, \ldots, N, \\
\frac{1}{\Delta t} M u_{n+1} &= \left(\frac{1}{\Delta t} M - V\left(0.5 u_n + 0.5 u^0_{n+1}\right) - D \right) u_n && \text{for } n \in 0, \ldots, N.
\end{aligned}
$$

This model was adjoined as a proof-of-concept for libadjoint. The author is unaware of any current free low-level AD tool that is applicable to this model for the reasons explained in section 1.3.2. First, work was undertaken to

implement the data callbacks, to allow libadjoint to manipulate Fluidity's scalar, vector and tensor function objects (section 2.5.3.2). Since the required functions are common operations in the context of finite-element discretisations, these were already implemented in Fluidity. Consequently, the data callbacks are simply thin wrapper functions to the appropriate Fluidity routines.

Next, a system was developed for model users to express their functional of interest. Fluidity embeds a Python interpreter (van Rossum et al., 2008) to provide dynamic programming facilities to model users. This was extended to provide a Python interface with which users could implement their own functionals of interest in a flexible and user-friendly manner. While this sacrifices some efficiency, these functionals are generally quite cheap to compute, and their cost is usually dominated by the cost of the forward and adjoint PDE solves.

Then, the model was annotated using the techniques presented in section 2.5.1. The library calls for the annotation are distributed through the model, close to the code which they describe: the annotation of each non-linear iteration happens within the non-linear iteration loop, the time step annotation happens within the time step loop, etc. At this point, the annotation was visualised in HTML (see figure 2.2) and inspected to ensure it matched the expectations of the model developer.

Once the annotation was complete, the operator callbacks were implemented and registered as described in section 2.5.3.1. For the Burgers' model the necessary operator callbacks are:

- an assembly and action callback for the identity operator I;
- an assembly callback for the term $\frac{1}{\Delta t} M$;
- an action callback for the timestepping operator T defined in (1.17);
- a callback providing the source terms for the replay test.

Matrices that do not change throughout the simulation, such as the mass and diffusion matrices, were cached. The advection matrix was reassembled at the arguments supplied. At this point, the annotation and callbacks were debugged by comparing the original model run against the forward replay mode of libadjoint (see section 2.5.6).

Finally, the necessary derivative callbacks were supplied to libadjoint. Following callbacks were required:

- an action callback computing the derivative of the non-linear advection operator V;
- a callback for the derivative of the functional of interest.

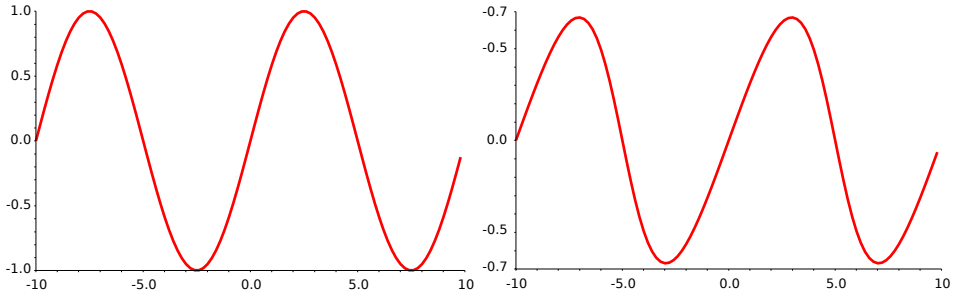
The differentiation of the functional of interest was achieved with *uncertainties*, an object-overloading automatic differentiation tool for Python (Lebigot, 2011). Differentiating the non-linear advection term was more challenging, as the assembly function made extensive use of derived types which are poorly supported by free low-level AD tools. Instead, the advection assembly subroutine was rewritten in Fortran 77, and code for its derivative was generated with TAPENADE, a source-to-source AD tool (Hascoët and Pascual, 2004). Again, it is to be emphasised that no freely-available AD tool is capable of differentiating the entire model, and so the pure low-level AD approach is impractical. However, it was entirely practical to reimplement a Fortran 77 version of just the non-linear advection operator, as the amount of code to convert to an AD-differentiable form was very small.

With the derivatives supplied, libadjoint successfully assembled each adjoint equation in turn, beginning at the end of simulation time and propagating backwards. The gradient of the functional with respect to various parameters (initial conditions, source terms) was computed with equation (1.11).

For verification and benchmarking, the Burgers' equation was solved in the domain interval $\Omega = [-10, 10]$ with initial condition $u(x) = \sin(x\pi/5)$, viscosity coefficient $\nu = 1$ and a time step size of $\Delta t = 1/32$ for $0 \leq t \leq 1$. The Burgers' equation was discretised in space using the finite element method with piecewise linear basis functions on a uniform mesh with $2 \cdot 10^4$ elements, and in time using the Crank-Nicolson scheme (Crank and Nicolson, 1947). The solutions of the forward model are shown in figure 2.5.

Next, the functional of interest and control parameters were specified. The functional was defined to be:

$$J(u, m) \coloneqq \int_\Omega \|u(t = 1)\|^2 \, \mathrm{d}x, \tag{2.18}$$



(a) $t = 0$ (b) $t = 1$

Figure 2.5.: The forward solution of the Burgers' example solved with Fluidity.

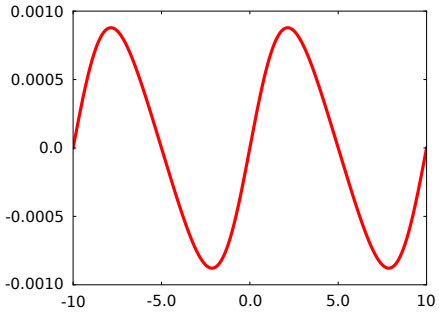


Figure 2.6.: The derivative of the functional of interest (2.18) with respect to the initial condition of the Burgers' example.

and the control parameter m was chosen to be the initial condition of u. The resulting functional gradient from the adjoint computation is shown in figure 2.6.

The correctness of the adjoint solution and the functional gradient was verified with the Taylor remainder convergence test (2.14). The results, shown in table 2.1, show the expected second order rate of convergence and give confidence in the correctness of the adjoint implementation.

Next, the efficiency of the implementation was benchmarked on one 2.13 GHz Intel Xeon CPU core with 12 GB memory. To obtain comparable timings for the forward and adjoint model without incorporating model specific overhead, the benchmark includes only the non-linear assembly, solver times and libadjoint related function calls. Specifically, the assembly of the linear operators is excluded, since they are assembled once in the initialisation step and then reused in both the forward and adjoint main loop. Moreover, the

h	$\lvert\tilde{J}(u_0 + h\delta u_0) - \tilde{J}(u_0)\rvert$	order	$\lvert\tilde{J}(u_0 + h\delta u_0) - \tilde{J}(u_0) - h\delta u_0^* \nabla\tilde{J}(u_0)\rvert$	order
$1 \cdot 10^{-4}$	$4.83 \cdot 10^{-6}$		$5.38 \cdot 10^{-8}$	
$5 \cdot 10^{-5}$	$2.40 \cdot 10^{-6}$	1.01	$1.35 \cdot 10^{-8}$	2.00
$2.5 \cdot 10^{-5}$	$1.20 \cdot 10^{-6}$	1.00	$3.63 \cdot 10^{-9}$	2.00
$1.25 \cdot 10^{-5}$	$5.98 \cdot 10^{-7}$	1.00	$8.40 \cdot 10^{-10}$	2.00

Table 2.1.: The Taylor remainders for the reduced functional $\tilde{J}$ (i.e. the functional of interest (2.18) considered as a pure function of the initial condition u_0) of the Burgers' example. The perturbation direction δu_0 is pseudorandomly generated. The expect order is one (two) for the remainder without (with) gradient information.

Number of mesh elements	$2 \cdot 10^4$	$4 \cdot 10^4$	$8 \cdot 10^4$	$16 \cdot 10^4$
Runtime of forward solve (s)	18	36	70	138
Runtime of adjoint solve (s)	20	39	78	157

Table 2.2.: Runtime comparison of the forward and adjoint Burgers' equation model with varying mesh resolution. The time step is fixed to $\Delta t = 1/32$.

runtime of the embedded Python interpreter and I/O are excluded. The averaged results of four runs are given in table 2.2. They show that the adjoint model is only marginally slower than the original forward code, even though the G^* contributions have to be assembled for the adjoint equations, which are not present in the forward run. For this example the adjoint implementation produced by libadjoint is therefore very efficient compared with the forward model.

2.7.2. Shallow water model

Fluidity also includes a shallow water model which is significantly more complex than the Burgers' equation solver presented in the previous section. The model solves the linear shallow water equations:

$$\frac{\partial u}{\partial t} + g\nabla\eta = 0,$$
$$\frac{\partial \eta}{\partial t} + H\nabla \cdot u = 0,$$

where u and η are the unknown velocity and free-surface displacement and the parameters g and H are the gravity constant and the free-surface elevation at rest, respectively. The equations are discretised with the $\mathrm{P1_{DG}}$-P2 finite element pair, which is both LBB-stable and can represent geostrophic balance exactly. A full description of the model equations and their discretisation can be found in Cotter et al. (2009). Unlike the Burgers' equation model, the shallow water model has more than one prognostic variable. The model is also made more complex by the fact that it solves on arbitrary one- or two-dimensional manifolds embedded in three-dimensional space, following the strategy of Bernard et al. (2009).

As the shallow water model is also part of Fluidity, it was possible to use the data callbacks for Fluidity's data types and the Python interface for implementing functionals from the previous work on the Burgers' equation model. The forward model was annotated, and the callbacks implemented. For this work, the adjoint of the linear shallow water model is chosen, as the implementation of the non-linear term on arbitrary manifolds is still under development. Since almost all of the operators were cached by the forward model anyhow, the callbacks merely used these cached matrices. The only matrices not cached were the projection operators associated with the capacity to solve on arbitrary manifolds. The code for these projection operators was modularised to isolate the functionality in subroutines that the callbacks could use.

The debugging features of section 2.5.6 were applied to detect potential implementation errors. The annotation of the forward model was verified by performing the forward replay test. Once this worked, the adjoint equations were assembled. An initial application of the Hermitian transpose consistency check (equation 2.12) indicated that the manifold projection operators were not self-adjoint. The required conjugate transposed projections were implemented by hand after which the Hermitian transpose consistency check passed. Finally, the correctness of the adjoint solution was verified using the Taylor remainder convergence test (2.14). Figure 2.3 shows the test results for the initial data assimilation configuration described in the next section. The Taylor remainder converged at second-order, giving confidence that the adjoint and gradient computations are correct.

h	$\lvert\tilde{J}(\eta^{est}+h\delta\eta^{est})-\tilde{J}(\eta^{est})\rvert$	order	$\lvert\tilde{J}(\eta^{est}+h\delta\eta^{est})-\tilde{J}(\eta^{est})-(h\delta\eta^{est})^{*}\nabla\tilde{J}(\eta^{est})\rvert$	order
$1\cdot10^{-4}$	$1.87\cdot10^{-5}$		$5.47\cdot10^{-10}$	
$5\cdot10^{-5}$	$9.33\cdot10^{-6}$	1.00	$1.37\cdot10^{-10}$	2.00
$2.5\cdot10^{-5}$	$4.67\cdot10^{-6}$	1.00	$3.42\cdot10^{-11}$	2.00
$1.25\cdot10^{-5}$	$2.33\cdot10^{-6}$	1.00	$8.55\cdot10^{-12}$	2.00

Table 2.3.: The Taylor remainders for the reduced functional $\tilde{J}$ (i.e. the functional of interest (2.19) considered as a pure function of the initial condition of η) of the shallow water example. The perturbation direction $\delta\eta^{est}$ is pseudorandomly generated. The expect order is one (two) for the remainder without (with) gradient information.

2.7.2.1. Application to an idealised data assimilation problem

To demonstrate the functionality and efficiency of the shallow water adjoint model it was used to solve an idealised data assimilation problem. In data assimilation, some of the parameters specifying the problem are not exactly known. However, observations of the solution (possibly at different time levels if the problem is time-dependent) are available. The goal of data assimilation is to find a better estimate of the unknown input parameters for which the solution best "fits" the observations, usually in a least-squares sense. These kind of problems arise in many fields of geosciences, in particular in weather prediction (Park and Xu, 2009).

This application is restricted to a simple shallow water setup in a two-dimensional, doubly-periodic domain $\Omega=[0,1]^2$. In the considered problem, the initial value (at time $t=0$) for the free-surface displacement η is unknown, while that of the velocity u is known. Given the observation of η at time $t=1/2$ and $t=1$ (an initial numerical experiment showed that the observation at $t=1/2$ is essential for successfully solving the problem), the initial condition of η that recovers these observations is sought. Therefore, the functional of interest is defined as:

$$J(\eta) \coloneqq \left\| \eta\left(t=\frac{1}{2}\right) - \eta^{\mathrm{obs}}_{t=\frac{1}{2}} \right\|^2_{L^2(\Omega)} + \left\| \eta(t=1) - \eta^{\mathrm{obs}}_{t=1} \right\|^2_{L^2(\Omega)}, \tag{2.19}$$

and the resulting optimisation problem is:

$$\min_{\eta^{\text{est}}} \; J(\eta^{\text{est}}) \quad \text{subject to} \tag{2.20a}$$

$$\frac{\partial u}{\partial t} + g\nabla\eta = 0, \tag{2.20b}$$

$$\frac{\partial \eta}{\partial t} + H\nabla \cdot u = 0, \tag{2.20c}$$

$$u(t=0) = u_0, \tag{2.20d}$$

$$\eta(t=0) = \eta^{\text{est}}, \tag{2.20e}$$

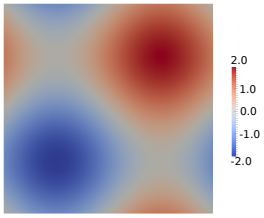
$$u(x=0) = u(x=1), \quad u(y=0) = u(y=1), \tag{2.20f}$$

$$\eta(x=0) = \eta(x=1), \quad \eta(y=0) = \eta(y=1). \tag{2.20g}$$

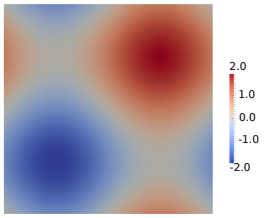
The minimum value of (2.20) is zero which is achieved if and only if the observations $\eta^{\text{obs}}_{t=1/2}$ and $\eta^{\text{obs}}_{t=1}$ are exactly recovered. The constraints (2.20b) and (2.20c) are the momentum and pressure equation of the linear shallow water equations. Equations (2.20d) and (2.20e) enforce the initial condition: u_0 is the known velocity initial condition and η^{est} is the estimate of the free-surface displacement initial condition. Finally, the equations (2.20f) and (2.20g) enforce periodic boundary conditions.

This optimisation problem can be solved iteratively as follows. First, use the estimate η^{est} to solve the shallow water model and its adjoint with respect to the misfit functional J. Then, equation (1.11) yields the derivative $dJ/d\eta$ at the current point in parameter space. Finally, a gradient-based optimisation algorithm is applied to obtain a better estimate for η^{est}, with which the procedure is restarted. For the following benchmarks, the L-BFGS-B method (Byrd et al., 1995) of SciPy (Jones et al., 2001) was used as optimisation algorithm. No bounds or memory limit were specified, in which case the L-BFGS-B method is equivalent to BFGS. Starting with an initial estimate $\eta^{\text{est}} = 0$, the optimisation loop was repeated until either the gradient norm of J was less than 10^{-12} or $(J^k - J^{k+1})/\max(|J^k|, |J^{k+1}|, 1) \leq \epsilon$, where the superscript denotes the iteration of the optimisation loop and ϵ is a small multiple of the machine precision.

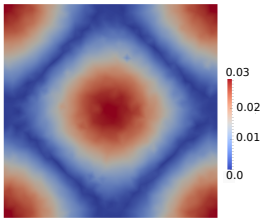
In order to verify the implementation of the solution procedure, its order of convergence was checked against theoretical results. For this, an analytical solution for problem (2.20) was constructed. By choosing $g = H = 1$, a



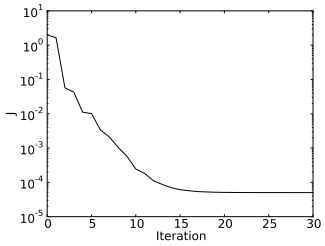
(a) Analytical initial layer thickness.



(b) Reconstructed initial layer thickness.



(c) Error in the reconstructed initial layer thickness.



(d) Optimisation convergence plot.

Figure 2.7.: The results of the shallow water data assimilation problem with analytical solution. The optimisation starts with a zero initial guess for the initial layer thickness and successfully reconstructed the analytical function. The remaining errors are mainly caused by errors in the wave speed introduced by the time discretisation, and can be reduced by choosing smaller time steps.

periodic solution for equations (2.20b) and (2.20c) is given by:

$$u^{\text{exact}} = \begin{pmatrix} \sin(2\pi(t+x)) \\ \sin(2\pi(t+y)) \end{pmatrix},$$
$$\eta^{\text{exact}} = -\sin(2\pi(t+y)) - \sin(2\pi(t+x)).$$

From this, the velocity initial condition and observations are derived:

$$u_0 = u^{\text{exact}}(t=0),$$
$$\eta^{\text{obs}}_{t=1/2} = \eta^{\text{exact}}(t=1/2),$$
$$\eta^{\text{obs}}_{t=1} = \eta^{\text{exact}}(t=1).$$

Mesh element size	0.3	0.3/2	0.3/4	0.3/8
L-BFGS-B iterations	24	27	29	31 + 9
Rate of convergence for η^{est}		2.0	2.3	2.1
Rate of convergence for $\eta(t = 1)$		1.6	2.2	2.2
Rate of convergence for $u(t = 1)$		2.3	2.2	2.3

Table 2.4.: Spatial order of convergence for the data assimilation problem. The time step size is $5 \cdot 10^{-5}$. The expected order is 2.

The numerical results of the data assimilation problem with these analytical observations on a mesh with element size 0.3/4 and a time step of 1/32 are shown in figure 2.7.

With a correct implementation, the solution of the optimisation problem is expected to converge to the analytical solution with the convergence rate given by the numerical analysis of the discretisation. The forward and adjoint shallow water problems were solved with the model described in section 2.7.2. The forward model uses Crank-Nicolson in time (Crank and Nicolson, 1947) and the P1_{DG}-P2 finite element in space (Cotter et al., 2009). It has been shown that this discretisation choice yields a second-order convergence in time and space (Cotter and Ham, 2011). However, any error in the implementation of the discretisation is likely to break either the spatial or temporal order of convergence. To derive the spatial order of convergence, the problem (2.20) was run with four different mesh resolutions and a small time step. The results are shown in table 2.4: as expected, second-order convergence is observed for both the control variable η^{est} and the final state variables (to observe second-order convergence for η^{est} with the highest resolution, it was necessary to perform 9 additional optimisation iterations after reaching the stopping criteria of the optimisation loop). The same technique was used to check the temporal order of convergence. The results in table 2.5 show again second order of convergence, giving high confidence in the correctness of the data assimilation implementation.

To benchmark the efficiency of the implementation, the average time required to solve the forward and adjoint system was recorded. The problem was solved four times and the timing results averaged. Table 2.6 shows almost identical timings for both the forward and adjoint solves, which can be interpreted as follows: consider the shallow water model cast in the form given in equation (2.1). The linearity of the problem yields a forward opera-

Time step (s)	1/4	1/8	1/16	1/32
L-BFGS-B iterations	20	20	30	28
Rate of convergence for η^{est}		1.9	1.9	2.0
Rate of convergence for $\eta(t=1)$		2.5	2.0	1.9
Rate of convergence for $u(t=1)$		1.9	1.9	2.0

Table 2.5.: Temporal order of convergence for the data assimilation problem. The mesh element size is 0.3/16. The expected order is 2.

Mesh element size	0.3/4	0.3/8	0.3/16
Time step (s)	1/32	1/64	1/128
Runtime of forward solve (s)	2.4	29	464
Runtime of adjoint solve (s)	2.5	29	474
libadjoint annotation time (s)	0.03	0.07	0.2

Table 2.6.: Averaged runtimes of the forward/adjoint solve and the total execution time of the libadjoint annotation in the data assimilation problem.

tor A which is independent of the solution vector. The definition of adjoint G operators (2.7) yields that $G = 0$. Similarly, the right hand side b of equation (2.1) does not depend on the solution, yielding $R = 0$. Hence the adjoint operator is just the Hermitian transpose of the forward operator A, see equation (2.6). An efficient adjoint implementation is therefore expected to be approximately as fast as the forward equivalent. It is remarkable that even though the application of libadjoint has been performed without any optimisation, the benchmark timings suggest that the efficiency of the resulting adjoint model is very close to this best case assumption. Finally, the execution time of libadjoint to annotate the forward model was measured and was found to be less than 2% of the forward running time, see table 2.6.

2.8. Summary

This chapter presented a library that facilitates and partly automates the development of adjoint models, based on the abstraction that the model is a sequence of equations solves. By annotating the forward model, libadjoint builds a symbolic description of the solved discrete system. With that, it can derive the symbolic description of the associated adjoint model. By

providing implementations of the operators in this description, libadjoint can then assemble and solve the adjoint equations.

The functionality was used to derive the adjoint of a Burgers' and shallow water model for which low-level AD tools are not directly applicable. In these examples, libadjoint only partly automated the adjoint model development, as the annotation and the implementation of the operators had to be performed manually. The next section demonstrates how these steps can be entirely automated by applying libadjoint to a high-level finite-element framework.

Chapter 3

Automated generation of adjoint models in FEniCS

Contents

3.1. Introduction **72**
3.2. The FEniCS system **74**
3.3. Applying libadjoint to DOLFIN **74**
3.4. Implementation **79**
3.5. Examples **82**
3.6. Summary **89**

This chapter derives and expands upon
P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 2012. submitted
The author was centrally involved in all aspects of the work.

Abstract

This chapter presents the automatic derivation and implementation of adjoint models for finite element models written in the FEniCS system (Logg et al., 2011; Logg, 2007). In FEniCS, the discrete problem is represented as high-level data from which a dedicated finite element compiler generates optimised parallel code. The adjoint derivation is achieved by symbolic manipulation via libadjoint as described in the previous chapter. All required tasks for applying libadjoint are automated by inspecting the high-level problem specification of the forward model. The resulting adjoint model is represented in the same high-level formulation as the forward model and so the same finite element compiler can be applied. As a consequence, the adjoint model implementation is efficient and runs naturally in parallel. Multiple examples demonstrate the automation, robustness and efficiency of this approach.

3.1. Introduction

The FEniCS system is a collection of software components for automating the solution of PDEs by the finite element method (Logg et al., 2011; Logg, 2007). The user specifies the variational form of the discrete equations in the domain specific language UFL (Alnæs, 2011), from which optimised finite element code is generated to compute their solutions. One of the key properties of this approach is that this high-level description can be algorithmically manipulated before the code generation is invoked. For example, the Newton solver in FEniCS makes use of this feature to form the Jacobian of the provided equations.

The main contribution of this chapter is to exploit the high-level representation of the discrete forward system to automatically derive the associated adjoint model. This is achieved by integrating libadjoint (see chapter 2) into DOLFIN (Logg and Wells, 2010; Logg et al., 2011), the user front end of FEniCS. The software library presented here, dolfin-adjoint, inspects the high-level problem description provided by the user at runtime, and performs the required tasks for applying libadjoint automatically. The re-

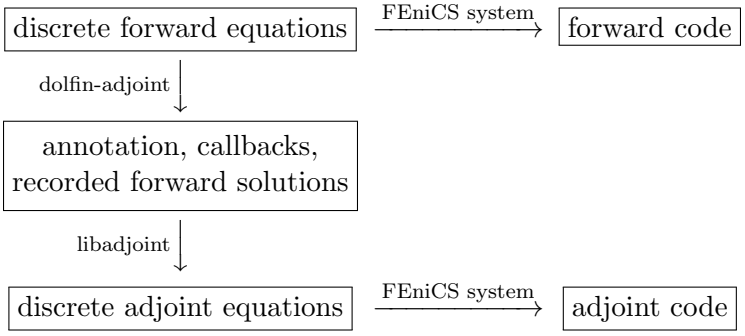


Figure 3.1.: The automated generation of the adjoint model in FEniCS. The user specifies the discrete forward equations in a high-level language similar to mathematical notation (UFL (Alnæs, 2011)). During the forward simulation, dolfin-adjoint creates the required annotation and operator callbacks. From that, libadjoint derives the corresponding representation of the discrete adjoint equations in UFL. Both the forward and adjoint equations are then passed to the code generator of the FEniCS system to compute the forward and adjoint solutions.

sulting adjoint model is represented in the same high-level data format as the forward model. Therefore, the code generation techniques in FEniCS can be applied to the adjoint model as easily as to the forward model, see figure 3.1.

Compared to applying libadjoint manually, this approach has the advantage that the derivation and solution of the adjoint model requires almost no user intervention. In addition, the adjoint derivation is generic: it applies to any PDE discretised with the finite element method, both to steady-state and time-dependent and both to linear and non-linear problems. Also, the adjoint model implementation is efficient and runs naturally in parallel, since it relies on the same code generation techniques as the forward model. Finally, the user has access to the checkpointing functionality and debugging tools implemented in libadjoint.

The finite element environment Sundance (Long et al., 2012) is similar to FEniCS in that it also operates on variational forms. In particular, it can automatically differentiate and adjoin individual variational forms. However, this automation does not extend to cases where the model consists

of a sequence of variational forms, as it is the typical case for time-dependent problems.

3.2. The FEniCS system

This section gives a brief introduction to the FEniCS system with the relevant details for the remaining sections of this chapter. A thorough overview can be found in Logg et al. (2011).

To solve a PDE with the FEniCS system, the user defines its discretised weak form in the domain specific language UFL (Alnæs, 2011), that mimics and encodes the mathematical formulation. This high-level formulation is then passed to a finite element form compiler such as FFC (Kirby and Logg, 2006), which generates optimised low-level code for the evaluation of the local element tensors. This generated code is used by DOLFIN to globally assemble and solve the problem. DOLFIN also provides the data structures for meshes, function spaces and functions. A list of commonly used DOLFIN functions is shown in table 3.1.

For time-dependent PDEs, the temporal discretisation is usually performed with a non-finite element discretisation scheme. In this case, the user writes the time loop manually and solves the variational problem for each time level as described above.

DOLFIN has interfaces for both C++ and Python. The Python interface uses just in time compilation, i.e. it invokes the code generation and compilation during runtime. In contrast, the code generation for the C++ interface happens as a preprocessing step before running the forward model. As a consequence, the high-level description of the forward problem is not directly available during runtime with the C++ interface. Because dolfin-adjoint relies on this data to perform runtime inspection and manipulation, the remaining sections discuss only the Python interface of DOLFIN.

3.3. Applying libadjoint to DOLFIN

This section presents the software package dolfin-adjoint which integrates DOLFIN with libadjoint. In particular, dolfin-adjoint performs the annotation of the forward model, the recording of the forward solutions, and the

DOLFIN function	Short description
`solve(a == b, u, bcs)`	Solve variational problem $a(u, v) = b(u)\ \forall v$
`u.assign(v)`	Copy function v to u
`assemble(a)`	Assemble variational form
`bc.apply(A)`	Apply a boundary condition to a matrix
`solve(A, x, b)`	Solve linear system
`project(u, V)`	Project u onto the function space V

Table 3.1.: Important DOLFIN statements relevant to dolfin-adjoint.

generation and registration of callback functions. All of these steps happen without any intervention by the model developer.

3.3.1. Annotation

With the annotation, libadjoint collects the necessary information about the discrete forward system to symbolically derive the associated adjoint system (see section 2.5.1). More specifically, each equation solve in the forward model must be annotated, with information about which variable the equation solves for and its (possibly non-linear) dependencies on previously computed forward solutions.

In dolfin-adjoint, the annotation happens automatically during the execution of the forward model. This is achieved by overloading all DOLFIN routines that modify the forward solution. These overloaded functions extract the required information from the UFL input and use the libadjoint interface to annotate the corresponding equation. It then calls the original DOLFIN routine and returns its result. All the DOLFIN routines listed in table 3.1 are overloaded by dolfin-adjoint.

As an example, consider the `solve` routine that is used to solve linear and non-linear variational forms. A linear variational problem $a(u, v) = b(v)\ \forall v$ may be solved by calling `solve(a == b, u)`, where `a` is a bilinear UFL form, `b` is a linear UFL form and `u` is the solution function. The overloaded `solve` routine annotates the corresponding linear equation as described in section 2.5.1. This annotation also contains information about the dependencies of operators in the equation on other forward variables, which dolfin-adjoint obtains by inspecting the associated UFL forms.

The `solve` function can also be used to solve non-linear equations of the form $F(u, v) = 0 \; \forall v$, where F is a linear UFL form in v but non-linear in u. Calling `solve(F == 0, u)` automatically invokes the Newton method to solve the non-linear system. Since libadjoint expects the annotation in the form of equation (2.1), dolfin-adjoint annotates the mathematically equivalent equation:

$$Iu = Iu - F(u), \tag{3.1}$$

where I represents the identity operator. Alternatively, each iteration in the Newton solve could be annotated. However, this approach would be computationally less efficient as it yields an adjoint equation for each iteration in the Newton solve. In contrast, annotating equation (3.1) results in only one adjoint equation from its linearisation (Gilbert, 1992).

Finally, the `solve` routine can be used to solve a linear system $Ax = b$ where A is an assembled matrix and b an assembled vector. Such preassembly is commonly applied to avoid repeatedly assembling the same discrete operator if it occurs multiple times in the forward model. In this case, the overloaded `solve` function cannot examine the dependencies of the operators on other forward variables, since A and b are not UFL forms, but low-level matrix/vector objects without semantic information. This problem is resolved by also overloading the `assemble` routine in order to associate every assembled object with its UFL form. The overloaded `solve` routine then uses this association to find the corresponding UFL form of the assembled objects. That way, it can access all the required information and perform the annotation as in the cases above.

DOLFIN provides alternative ways for solving variational forms by explicitly defining a solver object, such as `NewtonSolver`, `KrylovSolver` or `LUSolver`. dolfin-adjoint also overloads the solve routine of these classes and annotates the equations similar to the `solve` routine described above.

3.3.2. Recording the forward solutions

As described in section 2.5.2, libadjoint requires the forward variables in order to assemble the adjoint system. By default, no checkpointing scheme is used and dolfin-adjoint records all forward variables. Similar to the annotation, this step is performed in the overloaded DOLFIN routines: after

the original routine has been executed, the overloaded function records the solution as described in section 2.5.2 before returning the result.

Alternatively, the user may use a checkpointing scheme to reduce the storage requirement by recomputing some of the forward variables. For that, libadjoint requests information about which forward variables to include in a checkpoint from the forward model. The exact requirements were specified in definition 2 of section 2.6.2.2.

To be able to provide this information, dolfin-adjoint manages a list of forward variables that are currently in the program scope. When libadjoint requests the details about a checkpoint, dolfin-adjoint provides this list. It must be verified that this strategy fulfills the two requirements for a valid checkpoint given in definition 2. Firstly, the program scope obviously contains all forward variables that are required for future forward equations, and hence the first condition in definition 2 is satisfied. Secondly, the functionals of interest that are currently supported by dolfin-adjoint can all be written in the abstract form $J(u_1, u_2, ...) = \sum_{k \in K} j(u_k)$, where K denotes all or a subset of the time levels and u_k represents the forward variables of one time level. Therefore, the adjoint right hand side of the k'th adjoint equation $(\partial J/\partial u_k)^*$ only depends on u_k. Consequently, the second condition of definition 2 does not add further dependencies to the checkpoint.

3.3.3. Callbacks

As discussed in section 2.5.3, libadjoint requires certain function callbacks for the operators that occur in the annotation of the forward system. In dolfin-adjoint, these callbacks are created simultaneously with the annotation of the forward model: when an overloaded DOLFIN function annotates an equation, it also generates all required callbacks for the operators that occur in this equation.

The specific callbacks required depend on where the operator occurs in the annotation, see section 2.5.3. For example, if the operator is on the diagonal of the annotation, a callback for assembling its Hermitian transpose must be provided. Otherwise, a callback for computing the operator's Hermitian transpose action to a given vector is required. For any operator that has dependencies on forward solutions, libadjoint requires callbacks for computing its derivatives. To support the replay functionality (section 2.5.6), libadjoint

dolfin-adjoint statement	Short description
`J = Functional(inner(u, v)*dx*dt)`	Functional $\langle u, v\rangle_{\Omega\times(0,T)}$
`Functional(inner(u, v)*dx*dt[0:1])`	Functional $\langle u, v\rangle_{\Omega\times(0,1)}$
`Functional(inner(u, v)*dx*dt[1])`	Functional $\langle u(t=1), v(t=1)\rangle_{\Omega}$
`m = InitialConditionParameter(u)`	Parameter for the initial velocity
`compute_gradient(J, m)`	Computes the gradient $\mathrm{d}J/\mathrm{d}m$
`compute_adjoint(J)`	Generator for the adjoint solutions
`compute_tlm(m)`	Generator for the tangent linear solutions
`adj_checkpointing(...)`	Configure checkpointing

Table 3.2.: Important dolfin-adjoint statements. The notation $\langle u, v\rangle_{\Omega} \coloneqq \int_{\Omega} u \cdot v \,\mathrm{d}x$ is used for the L^2 inner product.

also requires callbacks for the non-Hermitian transpose operators and the right hand side of the forward model. In dolfin-adjoint, the operators are represented as UFL forms, which makes all these operations straightforward. For example, the adjoint variational form is obtained by swapping trial and test functions (Alnæs, 2011, §17.5.2). The generation of the derivatives employs the algorithmic differentiation capabilities of UFL (Alnæs, 2011, §17.7).

Besides these operator callbacks, libadjoint requires functions to perform algebraic operations such as addition, multiplication and solving variational problems. The implementation of these callbacks are thin wrappers around the corresponding DOLFIN functions and are independent of the specific forward model. In particular, the callback for solving equations consists of a thin wrapper around DOLFIN's `solve` function that invokes the code-generation for the assembly and solves the variational problem.

3.3.4. User interface

After loading the dolfin-adjoint module and executing the forward model, the user has multiple options to compute and use the adjoint solution. A list of important dolfin-adjoint statements is given in table 3.2. A high-level interface to dolfin-adjoint is `compute_gradient`: given a `Functional` J and a `Parameter` m, it computes the gradient $\mathrm{d}J/\mathrm{d}m$ with the adjoint approach. Alternatively, the adjoint solution can be accessed directly by using the `compute_adjoint` function. In addition, dolfin-adjoint supports

the computation of the tangent linear solution with `compute_tlm`, but is not further discussed here. A dolfin-adjoint implementation of the Burgers' problem and its adjoint is presented in figures 3.2 and 3.3.

libadjoint's checkpointing feature, described in section 2.6.2, is activated with the `adj_checkpointing` function, which sets the checkpointing strategy and the number of checkpoints that are available on disk and memory. dolfin-adjoint also interfaces to the debugging tools described in section 2.5.6.

Finally, dolfin-adjoint includes additional modules for advanced applications of the adjoint and tangent linear solutions, for example to perform stability analysis on the underlying problem. Another module, that has been developed as part of this thesis, allows the solution of PDE-constrained optimisation problems and will be the topic of chapter 4.

3.4. Implementation

dolfin-adjoint is written in the Python programming language. The source code and the presented examples are available at `http://dolfin-adjoint.org`.

3.4.1. Boundary conditions

Weakly enforced boundary conditions are included in the variational formulation and are handled implicitly by UFL's differentiation capabilities. Strongly enforced boundary conditions are passed as arguments to the `solve` routine which manipulates the discrete system accordingly. The `solve` routine overloaded by dolfin-adjoint inspects these boundary conditions, derives the associated boundary conditions for the adjoint system and includes these in the callback functions for libadjoint. In the case where the user manually assembles the discrete system, the strong boundary conditions are applied using the `apply` routine (see table 3.1). In this case, a similar strategy to the annotation of preassembled matrices (section 3.3.3) is used: dolfin-adjoint overloads the `apply` routine to create a mapping from the assembled object to its applied boundary conditions. This mapping is then used in the overloaded `solve` routine to access the high-level information about the bound-

```
1  from dolfin import *
2  # Overload the DOLFIN functions
3  from dolfin_adjoint import *
4  # Define the mesh and function spaces
5  n = 10000
6  mesh = UnitInterval(n)
7  V = FunctionSpace(mesh, "CG", 2)
8  u = project(Expression("sin(2*pi*x[0])"), V)
9  u_next = Function(V)
10 v = TestFunction(V)
11 nu = Constant(0.01)
12 timestep = Constant(0.05)
13 # Define the weak form of the Burgers' equation
14 F = ((u_next - u)/timestep*v
15      + u_next*grad(u_next)*v + nu*grad(u_next)*grad(v))*dx
16 bc = DirichletBC(V, 0.0, "on_boundary")
17 # Define the simulation period
18 t = 0.0; T = 2.0
19 adjointer.time.start(0)
20 # Perform the time loop
21 while (t <= T):
22     solve(F == 0, u_next, bc)
23     u.assign(u_next)
24     # Update the simulation time
25     t += float(timestep)
26     adj_inc_timestep(time = t, finished = t>T)
```

Figure 3.2.: DOLFIN code for a simple discretisation of the Burgers' equation (1.12) with dolfin-adjoint. The dolfin-adjoint module overloads the existing `solve` and `assign` functions. The only changes for applying dolfin-adjoint are the inclusion of the module (line 3) and lines 19 and 26 to provide the necessary time step information for libadjoint.

```
1 J = Functional(0.5*inner(u, u)*dx*dt)
2 m = InitialConditionParameter(u)
3 dJdm = compute_gradient(J, m)
```

Figure 3.3.: Sample dolfin-adjoint user code complementing the Burgers' model presented in figure 3.2: the adjoint model is generated and used to compute the gradient of the functional $J = \frac{1}{2}\langle u, u\rangle_{\Omega\times(0,T)}$ with respect to the initial condition $u(t=0)$.

ary conditions and to apply the associated adjoint boundary conditions in the same way as before.

3.4.2. Parallelism

One of the advantages of libadjoint is that it works on a high-level abstraction that lies above the implementation details. In particular, the annotation is independent of the parallel execution of the model, as the equations to be solved do not depend on how the computation is distributed. Instead, libadjoint leaves it to the callbacks to handle the distributed computation.

Similarly, the description of the forward model in the UFL form language does not contain parallel specific information. Instead, DOLFIN derives parallel communication patterns and their implementation automatically. Since dolfin-adjoint operates on the level of the UFL form language, the entire derivation of the adjoint system is performed without parallel specific code. Instead, the parallel communication calls for the adjoint model are automatically obtained in exactly the same way as for the forward model. This works both for the MPI and OpenMP cases (Logg et al., 2011, §6.4).

3.4.3. Limitations

While the derivation of the adjoint model with dolfin-adjoint works in many cases without user intervention, there are some limitations to be considered.

An obvious mathematical requirement is that the forward model must be differentiable. Non-differentiable operators in the forward model can either originate from the continuous PDE, or can be introduced during the discretisation step (for example with upwind schemes (Liu and Sandu, 2005)). Furthermore, adaptive discretisation methods that dynamically vary the spatial or temporal resolution to minimise the discretisation error are usually non-differentiable. In these cases, the resulting gradient information will not be consistent with the discrete forward model. Nevertheless, it may provide enough information for certain applications such as PDE-constrained optimisation. Alternatively, the non-differentiable operators can be replaced by smooth approximations. This strategy will be used in chapter 6.

Another limitation is, that not all relevant DOLFIN functions are overloaded by dolfin-adjoint, such as functions to access DOLFIN's low-level data structures directly. If such a non-overloaded function is used, the re-

sulting annotation, and hence the computed gradient, is inconsistent with the forward model. However, this inconsistency can easily be detected by running the replay debugging tool provided by libadjoint (section 2.5.6).

3.5. Examples

This section presents three numerical examples to demonstrate the capabilities of dolfin-adjoint. All experiments were performed on 2.13 GHz Intel Xeon CPU cores with 12 GB memory. The presented timing benchmarks are averaged results of five runs. More examples can be found in Farrell et al. (2012) and the remaining chapters of this thesis.

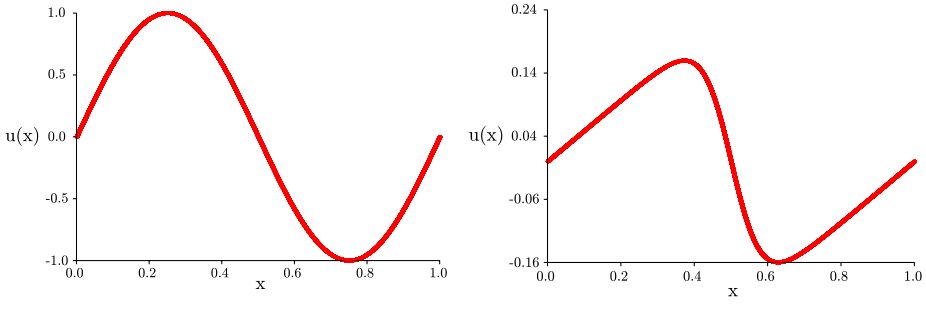
3.5.1. Burgers' equation

The first example considers the implementation of the viscous Burgers' problem using the implementation shown in figure 3.2. It solves the Burgers' equation (1.12) on the unit interval with homogeneous Dirichlet boundary conditions and a viscosity coefficient of $\nu = 0.01$. The result of the forward model is shown in figure 3.4.

To test the adjoint model, the functional of interest was defined to compute:

$$J(u) = \frac{1}{2} \langle u, u \rangle_{\Omega \times (0,T)} , \tag{3.2}$$

with a final time of $T = 2$. The required code changes to compute the gradient information with the adjoint approach is shown in figure 3.3. The



(a) $t = 0$ (b) $t = 2$

Figure 3.4.: The forward solution of the Burgers' example.

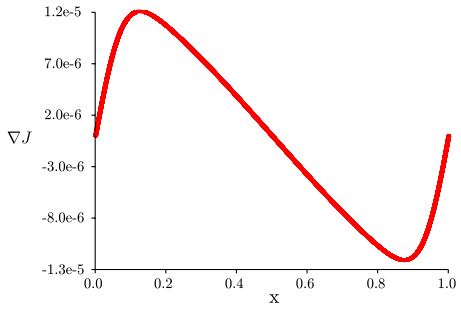


Figure 3.5.: The derivative of the functional of interest (3.2) with respect to the initial condition of the Burgers' example.

h	$\lvert\tilde{J}(u_0 + h\delta u_0) - \tilde{J}(u_0)\rvert$	order	$\lvert\tilde{J}(u_0 + h\delta u_0) - \tilde{J}(u_0) - h\delta u_0^* \nabla\tilde{J}(u_0)\rvert$	order
$1 \cdot 10^{-4}$	$1.89 \cdot 10^{-8}$		$2.11 \cdot 10^{-9}$	
$5 \cdot 10^{-5}$	$8.90 \cdot 10^{-9}$	1.08	$5.28 \cdot 10^{-10}$	2.00
$2.5 \cdot 10^{-5}$	$4.32 \cdot 10^{-9}$	1.04	$1.32 \cdot 10^{-10}$	2.00
$1.25 \cdot 10^{-5}$	$2.13 \cdot 10^{-9}$	1.02	$3.30 \cdot 10^{-11}$	2.00
$6.25 \cdot 10^{-6}$	$1.06 \cdot 10^{-9}$	1.01	$8.26 \cdot 10^{-11}$	2.00

Table 3.3.: The Taylor remainders for the reduced functional $\tilde{J}$ (i.e. the functional of interest (3.2) considered as a pure function of the initial condition) of the Burgers' example. The perturbation direction δu_0 is pseudorandomly generated. The expect order is one (two) for the remainder without (with) gradient information.

resulting functional gradient with respect to the initial condition is presented in figure 3.5.

In order to verify the correctness of the adjoint model, the Taylor remainder convergence test (section 2.5.6) was applied. The results are presented in figure 3.3 and show the expected first and second order of convergence.

The following table shows a breakdown of the simulation runtimes:

	Runtime (s)	Runtime (%)
Forward model	10.97	64.1
Annotation	0.26	1.5
Adjoint model	5.88	34.4

The Newton solver of the forward model converged after 2 to 3 iterations. Since each Newton solve corresponds to one linear solve in the adjoint model,

an efficient adjoint implementation is expected to require approximately half the time of the forward model, which is also observed from the timings above.

3.5.2. Navier-Stokes equations

The next example solves the adjoint problem of the steady-state Navier-Stokes equations:

$$u \cdot \nabla u - \nu \nabla^2 u + \nabla p = 0,$$
$$\nabla \cdot u = s,$$

where u is the unknown velocity, p the unknown pressure, ν the viscosity coefficient and s the pressure source term.

The considered setup is based on a demonstration example that is included in DOLFIN. The two-dimensional domain consists of a rectangle of size 4×1 with a rectangular shaped obstacle in the center of the domain. The viscosity coefficient is set to $\nu = 0.02$ and the pressure is set to one on the left boundary and zero on the right boundary. A no-slip condition is enforced on the remaining boundaries.

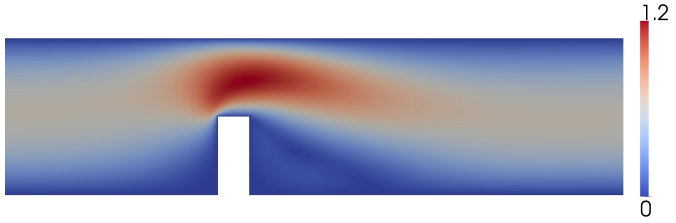
The mesh is structured and consists of $12,480$ triangular elements. The Navier-Stokes equations are discretised with the P2-P1 Taylor-Hood finite element pair and solved with a Newton method. The results of the forward model are shown in figure 3.6.

The functional of interest is defined as:

$$J(u) = \frac{1}{2} \langle u, u \rangle_\Omega \,. \tag{3.3}$$

The only necessary change to obtain the adjoint model was to include the dolfin-adjoint module. The functional gradient with respect to the pressure source $\mathrm{d}J/\mathrm{d}s$ is shown in figure 3.7.

To verify the gradient computation, the Taylor remainder convergence test was performed. The results are shown in table 3.4. The Taylor remainders converge at the expected order, indicating that the functional gradient computed using the adjoint is correct.



(a) Velocity



(b) Pressure

Figure 3.6.: The forward solutions of the Navier-Stokes example.

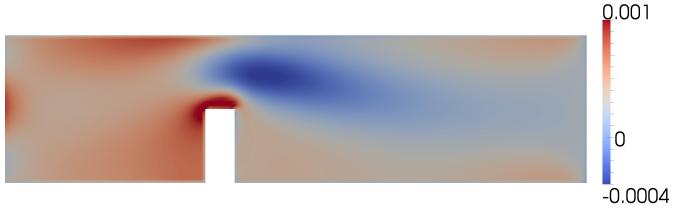


Figure 3.7.: The derivative of the functional of interest (3.3) with respect to the pressure source s of the Navier-Stokes example.

h	$\lvert\tilde{J}(s+h\delta s)-\tilde{J}(s)\rvert$	order	$\lvert\tilde{J}(s+h\delta s)-\tilde{J}(s)$ $-h\delta s^*\nabla\tilde{J}(s)\rvert$	order
$1\cdot10^{-5}$	$9.87\cdot10^{-6}$		$2.1908\cdot10^{-10}$	
$5\cdot10^{-6}$	$4.94\cdot10^{-6}$	1.00	$5.48\cdot10^{-11}$	2.00
$2.5\cdot10^{-6}$	$2.47\cdot10^{-6}$	1.00	$1.37\cdot10^{-11}$	2.00
$1.25\cdot10^{-6}$	$1.23\cdot10^{-6}$	1.00	$3.43\cdot10^{-12}$	2.00
$6.25\cdot10^{-7}$	$6.17\cdot10^{-7}$	1.00	$8.54\cdot10^{-13}$	2.00

Table 3.4.: The Taylor remainders for the reduced functional $\tilde{J}$, equation (3.3) for the Navier-Stokes example. The perturbation direction δs is pseudorandomly generated. The expect order is one (two) for the remainder without (with) gradient information.

The timing results for this example are:

	Runtime (s)	Runtime (%)
Forward model	15.75	81.1
Annotation	< 0.01	< 0.1
Adjoint model	3.67	18.9

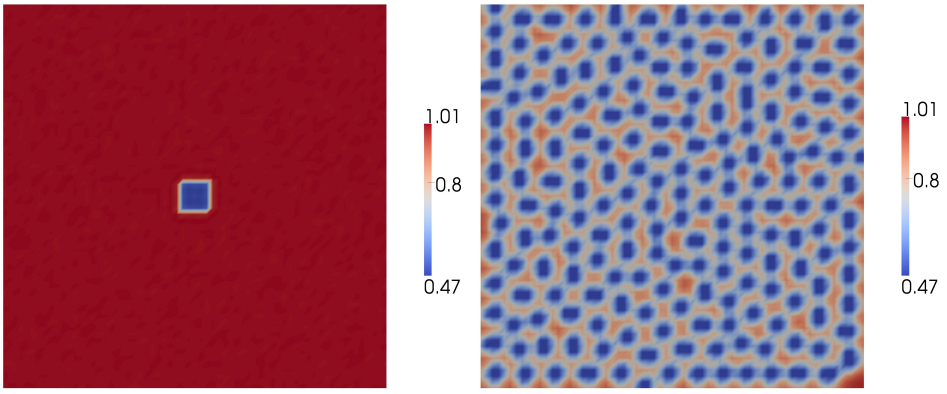
The time required for the annotation was very small and below the timing variations of the experiments performed. The Newton solver requires five iteration to converge, and therefore the optimal relative runtime of the adjoint model is $1/6 \approx 17\%$. The measured value of 18.9% indicates that the adjoint model implementation is indeed efficient.

3.5.3. Gray-Scott equations

The final example considers the numerical simulation of the Gray-Scott equations (Gray and Scott, 1983). They describe a simple reaction-diffusion problem which exhibits an extraordinary range of patterns, including stripes, spots, rolls, waves, many of which mimic patterns found in nature (Pearson, 1993). The Gray-Scott equations are:

$$\begin{aligned} \frac{\partial u}{\partial t} &= D_u \nabla^2 u - uv^2 + F(1-u), \\ \frac{\partial v}{\partial t} &= D_v \nabla^2 v + uv^2 - (F+k)v, \end{aligned} \tag{3.4}$$

where u and v are the two unknowns, D_u and D_v are the diffusion coefficients, k is the rate constant of the second reaction and F the feed rate. The setup considered here was adopted from Pearson (1993). The two-dimensional domain consists of a square of dimension $\Omega = [0.0, 2.5]^2$ and periodic boundary conditions. Following Pearson (1993), the initial conditions are constructed as follows. First, u and v are set to 1 and 0, respectively. Then, the values inside the centered square with length 0.1 were perturbed to $1/2$ for u and $1/4$ for v. Finally, these conditions are perturbed with a $\pm 1\%$ pseudorandomly generated noise to break the square symmetry. The diffusion coefficients are set to $D_u = 2 \cdot 10^{-5}$ and $D_v = 10^{-5}$. The solution patterns highly depend on the rate constant k and the feed



(a) $T = 0$ (b) $T = 20,000$

Figure 3.8.: The forward solution u of the Gray-Scott equations. The initial perturbation propagates slowly over the domain and reaches a quasi steady-state.

rate F. The experiments presented here are performed with $k = 0.055$ and $F = 0.02$, for which Pearson (1993) observed a dotted solution.

The Gray-Scott equations (3.4) are solved using piecewise linear functions in space and the explicit Euler method in time with a timestep of 1. The initial perturbation slowly spreads over the domain and reaches a quasi-steady state at approximately $T = 20,000$, see figure 3.8.

The functional of interest is defined as:

$$J(u) = \langle u(t = T), u(t = T) \rangle_\Omega \,. \tag{3.5}$$

At this point, the `compute_gradient` routine could be used without any further changes to compute the functional gradient with respect to the initial value of u using the adjoint approach. By default, dolfin-adjoint stores every forward solution in memory for the adjoint computation. An initial test revealed that storing the forward solutions of the $20,000$ time levels exceeded the available computer memory. Therefore, libadjoint's checkpointing functionality was set up to use 100 checkpoints in memory and on disk. This reduced the required amount of memory drastically and made it possible to complete the computation. The result, performed in parallel on 4 CPUs, is presented in figure 3.9. In total, less than 20 lines of code in the forward model had to be changed to obtain these results.

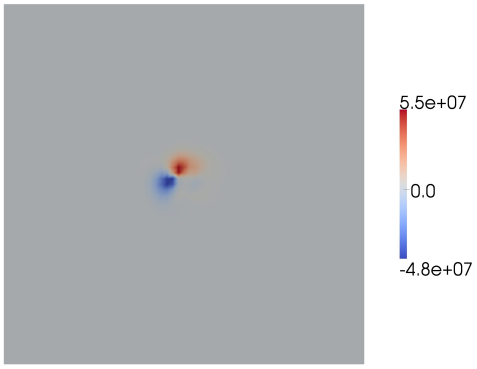


Figure 3.9.: The functional gradient for the Gray-Scott example, computed with the adjoint approach. The plot shows the sensitivity of the functional of interest (3.5) with respect to the initial value of u.

h	$\lvert\tilde{J}(u_0 + h\delta u_0) - \tilde{J}(u_0)\rvert$	order	$\lvert\tilde{J}(u_0 + h\delta u_0) - \tilde{J}(u_0) - h\delta u_0^* \nabla\tilde{J}(u_0)\rvert$	order
$1 \cdot 10^{-2}$	$3.25 \cdot 10^{-5}$		$6.15 \cdot 10^{-8}$	
$5 \cdot 10^{-3}$	$1.63 \cdot 10^{-5}$	1.00	$1.54 \cdot 10^{-8}$	2.00
$2.5 \cdot 10^{-3}$	$8.14 \cdot 10^{-6}$	1.00	$3.84 \cdot 10^{-9}$	2.00
$1.25 \cdot 10^{-3}$	$4.07 \cdot 10^{-6}$	1.00	$9.61 \cdot 10^{-10}$	2.00
$6.25 \cdot 10^{-4}$	$2.03 \cdot 10^{-6}$	1.00	$2.40 \cdot 10^{-10}$	2.00

Table 3.5.: The Taylor remainders for the Gray-Scott example. The perturbation direction δu_0 is pseudorandomly generated. The expect order is one (two) for the remainder without (with) gradient information.

To verify the correctness of the adjoint solution, the Taylor remainder convergence test described in section (2.5.6) was applied. To reduce the computational cost of the following tests, the final time was set to $T = 100$. The results of the Taylor test are listed in table 3.5. As expected, the first and second convergence orders hold, indicating that the computed gradient is correct.

To investigate the performance of the checkpointing scheme, the simulation was repeated with a final time of $T = 100$, 7 checkpoints (plus one additional checkpoint for libadjoint's internal use, see section 2.6.2) in memory and 0 checkpoints on disk. An initial upper limit for the expected computational cost of the recomputations can be determined with inequal-

ity (2.16). The inequality is fulfilled for $t = 3$, and hence the runtime of (re-)computing forward variables should be less than three times the cost of the forward model. As can be seen from the following table, the optimal checkpointing scheme predicts a slowdown ratio of only 2.55:

	Advance steps	Runtime of advance steps
Without checkpointing	100	20.73 s
With checkpointing	255	53.95 s
Ratio	2.55	2.60

The measured slowdown ratio evaluates to 2.6 and hence confirms this prediction.

Finally, the breakdown of the runtimes without checkpointing is:

	Runtime (s)	Runtime (%)
Forward model	19.69	44.8
Annotation	1.04	2.4
Adjoint model	23.21	52.8

The relatively large annotation time compared to the previous examples is caused by the explicit temporal discretisation. It requires many computationally cheap time step computations, while the cost of annotation stays constant. Therefore, a larger proportion of time is spent in the annotation, which has not been optimised. The Newton solver converges in most cases after a single iteration and therefore an efficient adjoint system computation should take approximately as long as the forward model. This expectation is confirmed by the runtime results.

3.6. Summary

In this chapter, libadjoint has been applied to DOLFIN, a high-level framework for solving PDEs. With this integration, the adjoint model of a forward model written in the Python interface of DOLFIN can in many cases be derived fully automatically. This is achieved by overloading the DOLFIN routines that are relevant for automatically applying libadjoint. In particular, this includes building the annotation, recording the forward solutions

and generating the function callbacks. Necessary derivative operators are obtained with the differentiation capabilities of UFL. The discrete adjoint model so derived is represented in the same high-level language as the forward model. Therefore, the code generation techniques of FEniCS can be applied in the same way for both the forward and adjoint system. The resulting adjoint model implementation is efficient and runs naturally in parallel. In addition, the advanced features of libadjoint, such as checkpointing and debugging tools, are easily accessible to the model developer.

An extension module of dolfin-adjoint to solve PDE-constrained optimisation problems is the topic of the next chapter.

Chapter 4

A framework for PDE-constrained optimisation

Contents

4.1. Introduction 92
4.2. The reduced optimisation problem 93
4.3. Optimisation algorithms 96
4.4. The optimisation framework 101
4.5. Examples 104
4.6. Verification 110
4.7. Summary 114

Abstract

This chapter presents a generic framework for solving PDE-constrained optimisation problems. The governing PDE is defined in a high-level language using the FEniCS framework. A wide range of PDEs are supported, including transient, non-linear and coupled systems. From the high-level problem specification, the framework automatically generates the required interfaces to various gradient-free and gradient-based optimisation algorithms. The derivative information for gradient-based algorithms is computed with the adjoint approach using dolfin-adjoint.

Examples on classical problems show the flexibility of this approach, followed by solution of novel optimisation problems in the next chapters.

4.1. Introduction

PDE-constrained optimisation has applications in many areas of science and engineering such as parameter estimation (Navon, 1998), data assimilation (Le Dimet and Talagrand, 1986), optimal design (Giles and Pierce, 2000) and optimal control (Lions, 1971). The main contribution of this chapter is the development of an extension module to dolfin-adjoint, that allows the user to define and solve such optimisation problems in a generic, flexible and efficient way. The user specifies the discretised governing PDE, the functional of interest and the optimisation parameters in the finite element framework FEniCS (section 3.2) and the user interface to dolfin-adjoint (section 3.3.4). With that, the presented module automatically performs all necessary steps to minimise or maximise the functional of interest using the given parameters.

The optimisation process consists of iteratively evaluating the functional of interest and its gradient at different points in parameter space. The key idea of this chapter is to automate these evaluations by operating purely on the tape that is recorded by libadjoint. In particular, a functional evaluation is obtained by replaying the tape (which runs the forward model). The functional gradient is obtained by deriving and solving the adjoint model as described in the previous chapters. If the optimisation algorithm updates the point in parameter space, the tape is modified accordingly to reflect these changes.

This approach adopts many of the advantages from the FEniCS system and dolfin-adjoint. The user can formulate the problem in the UFL form language. The presented optimisation module can be applied to linear and non-linear, as well as steady-state and time-dependent governing PDEs. It also interfaces to various optimisation algorithms, including local gradient-based and global gradient-free methods. Finally, the governing PDEs can be solved in parallel, which is often crucial for reasonable runtimes.

Multiple alternative frameworks for PDE-constrained optimisation exist. van Bloemen Waanders et al. (2002) developed an optimisation framework based on the finite-element software Sundance. However the build-in automatic adjoint derivation of Sundance does not extend to cases where the problem consists of a sequence of variational problems, which is the typically the case for time-dependent problems. The Stanford University Unstructured suite (Palacios et al., 2012) and RoDoBo (Becker et al., 2005) provide platforms that combine necessary tools for solving PDE-constrained optimisation problems. In contrast to the approach taken here, these software packages compute the gradients using the continuous adjoint approach and require the manual derivation and implementation of the adjoint model.

4.2. The reduced optimisation problem

This chapter considers an extended version of the general optimisation problem (1.1):

$$\begin{aligned} \min_{u,m} \; & J(u,m) \\ \text{subject to } & F(u,m) = 0, \\ & h(m) = 0, \\ & g(m) \leq 0, \end{aligned} \tag{4.1}$$

where m contains the optimisation parameters (also called control parameters for optimal control problems), $F(u,m) = 0$ is a system of partial differential equations parameterised by m with solution u, and $J(u,m) \in \mathbb{R}$ is the functional of interest that is to be minimised. The equality and inequality constraints $h = 0$ and $g \leq 0$ enforce additional conditions on the optimisation parameter m. A common example is a box constraint of the form:

$$a \leq m \leq b. \tag{4.2}$$

In addition to the regularity assumptions made in section 1.1.4, it is assumed that g and h are continuously Fréchet differentiable.

Figure 4.1 shows two common ways for solving such optimisation problems, and the approach taken here. One possibility is to apply optimisation algorithms directly to the discretised problem (4.1). Here, an alternative formulation is considered, where the functional of interest J is replaced

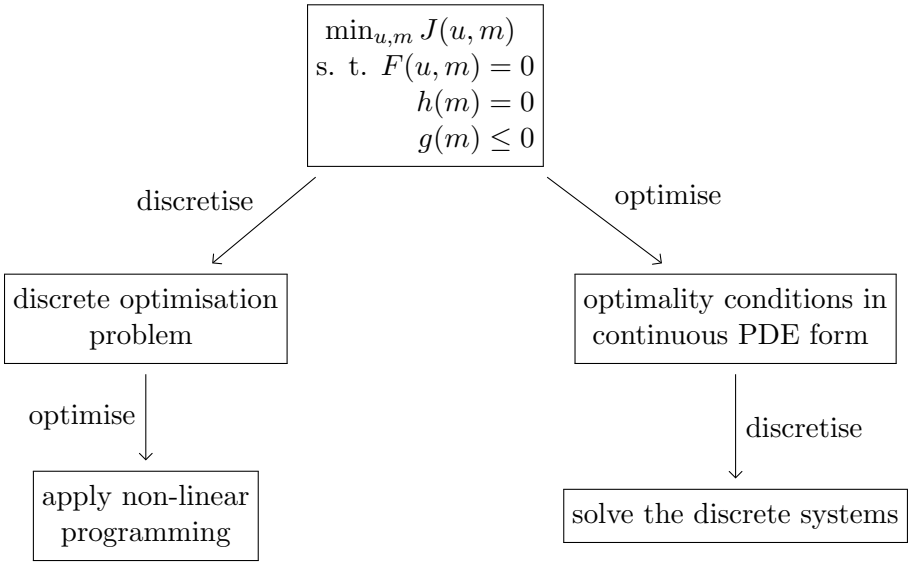


Figure 4.1.: Two common approaches for solving PDE-constrained optimisation problems. The left path, known as discretise-then-optimise, first discretises the problem. The resulting finite-dimensional optimisation problem and can then be solved using appropriate solution algorithms. The right path, known as optimise-then-discretise, first derives the optimality conditions on the continuous level and then discretises and solves these. Both approaches yield the adjoint equation as part of the optimality condition. This chapter focuses on the discretise-then-optimise strategy.

with the reduced functional (1.4). Since the reduced functional implicitly enforces the solution of the PDE, the PDE-constraint becomes superficial. The result is the *reduced optimisation problem*:

$$\begin{aligned} &\min_m \tilde{J}(m) \\ \text{subject to } &h(m) = 0, \\ &g(m) \leq 0. \end{aligned} \tag{4.3}$$

Both formulations (4.1) and (4.3) have advantages and disadvantages. The unreduced formulation allows the natural treatment of constraints for the PDE solution u, e.g. a box constraint of the form $a \leq u \leq b$. In addition, the optimisation iterations are typically computationally cheaper, as many optimisation algorithms linearise the constraint equations, and hence only linear PDEs must be solved, even if the governing PDE is non-linear.

A potential downside of the unreduced formulation is that both the PDE solution u and the optimisation parameter m are free variables that are to be optimised. Consequently, the number of free variables can be millions or billions for a time-dependent large scale governing PDE. Many modern optimisation algorithms do not scale to problems of this size, and hence specially designed algorithms must be used instead. Furthermore, the current estimate of the whole PDE solution u must be stored, which can be prohibitive for large scale time-dependent governing PDEs. In contrast, the reduced formulation contains only the optimisation parameter m as a free variable, which is typically of much smaller dimension than the PDE solution u. Another advantage of the reduced formulation is that the governing PDE is satisfied after each optimisation iteration, since it is implicitly solved for in the reduced functional. Hence, the optimisation loop can be stopped as soon as the functional is sufficiently reduced.

The following example illustrates these formulations on a classical optimal control problem (see for example Tröltzsch (2005, chapter 2.1.5) or Hinze et al. (2009, chapter 1.5.3)): given a thin, heatable plate with fixed temperature at the boundary, what is the optimal cooling or heating function in order to obtain a desired temperature profile across the plate? This problem can be formulated as an optimisation problem constrained by the stationary heat equation:

$$\begin{aligned} \min_{u,m} \ & \frac{1}{2}\|u - u_d\|^2_{L^2(\Omega)} + \frac{\alpha}{2}\|m\|^2_{L^2(\Omega)} \\ \text{subject to} \ & -\nabla^2 u = m && \text{on } \Omega, \\ & u = 0 && \text{on } \partial\Omega, \\ & a \leq m \leq b && \text{on } \Omega. \end{aligned} \tag{4.4}$$

Here u_d is the desired temperature profile and u is the solution of the stationary heat equation with homogeneous Dirichlet boundary conditions. The functional of interest measures the error between the PDE solution and the desired temperature profile plus a regularisation term that is multiplied by a scaling factor $\alpha \geq 0$. The optimisation parameter m controls the heat source and is limited by the values in the box constraints.

Under mild assumptions, the heat equation with Dirichlet boundary conditions yields a unique solution for any source parameter m (Hinze et al., 2009, §1.3.1.1). Hence there exists a solution operator $u(m)$ and the reduced problem may be formulated:

$$\begin{aligned} \min_{m} \ & \frac{1}{2}\|u(m) - u_d\|^2_{L^2(\Omega)} + \frac{\alpha}{2}\|m\|^2_{L^2(\Omega)} \\ \text{subject to} \ & a \leq m \leq b && \text{on } \Omega. \end{aligned}$$

4.3. Optimisation algorithms

This section introduces two common gradient-based algorithms for non-linear optimisation that are suitable to solve the reduced problem (4.3): an extension to the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method and sequential quadratic programming (SQP). Both these algorithms will be used in the upcoming chapters of this thesis. A detailed overview on optimisation algorithms can be found in Nocedal and Wright (1999); Wang et al. (2010) and a mathematical discussion of PDE-constrained optimisation is given by Hinze et al. (2009).

4.3.1. Broyden-Fletcher-Goldfarb-Shanno method

The BFGS method is a widely used algorithm for solving unconstrained, non-linear optimisation problems. For its derivation, consider the reduced optimisation problem (4.3) without any constraints:

$$\min_m \tilde{J}(m). \tag{4.5}$$

The first-order optimality conditions of problem (4.5) demand that the functional gradient vanishes at every local minimum m^{opt}:

$$\nabla \tilde{J}\left(m^{\text{opt}}\right) = 0. \tag{4.6}$$

One way of solving equation (4.6) is to apply the Newton method (assuming that the second derivatives of the reduced functional exist). Starting with an initial guess $m^{(0)}$, a sequence of optimisation variables $m^{(1)}, m^{(2)}, \ldots$ is computed using the Newton update:

$$m^{(k+1)} = m^{(k)} - \left(\mathcal{H}\tilde{J}\left(m^{(k)}\right)\right)^{-1} \nabla \tilde{J}\left(m^{(k)}\right), \tag{4.7}$$

where $\mathcal{H}$ denotes the Hessian operator.

While the Newton method benefits from a fast local convergence, it may not converge if the initial guess is too far away from a solution m^{opt}. This issue can be overcome by globalisation strategies such as line searches (Nocedal and Wright, 1999, §3) or the trust region method (Nocedal and Wright, 1999, §4).

One of the main drawbacks of the Newton method is that the Newton update (4.7) requires the second-order derivatives and the solution of a linear system with the Hessian operator. Because the Hessian matrix is usually dense, the usage of the full Hessian in the Newton update has traditionally been viewed as too computationally expensive for large scale optimisation problems, which led to development of alternatives methods such as quasi-Newton methods (Dennis and Moré, 1977). Nevertheless, second-order adjoints can be used to compute second-order derivatives (section 1.2.4), and have been used in combination with Hessian based optimisation methods (Guillaume and Masmoudi, 1994; Raffard and Tomlin, 2005; Cioaca et al., 2012).

Quasi-Newton methods circumvent the need for second derivatives by replacing the Hessian matrix with an approximation B_k in the Newton update (4.7), that is:

$$m^{(k+1)} = m^{(k)} - B_k^{-1} \nabla \tilde{J}\left(m^{(k)}\right). \tag{4.8}$$

ALGORITHM 4: Quasi-Newton algorithm

Data: initial optimisation variable m^0
Result: optimal value for m
$k \leftarrow 1$
set B_1 to a symmetric, positive definite matrix
repeat
  compute quasi-Newton direction $s \leftarrow -B_k^{-1} \nabla \tilde{J}(m^{(k)})$
  determine step size $t > 0$ by performing a line search
  $m^{(k+1)} \leftarrow m^{(k)} + ts$
  compute B_{k+1}
  $k \leftarrow k + 1$
until *stopping criteria*

An outline of the quasi-Newton algorithm with line searches is given in algorithm 4. The determination of a suitable step size with the line search ensures the global convergence of the algorithm.

There are different strategies for how to compute the Hessian approximations B_k in the quasi-Newton update (4.8). One of the most popular approaches is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, which was independently proposed by Broyden (1970); Fletcher (1970); Goldfarb (1970) and Shanno (1970). Starting with an initial approximation B_1, it computes the new Hessian approximation by the following rank-two update:

$$B_{k+1} = B_k + \frac{yy^*}{y^*s} - \frac{B_k s s^* B_k}{s^* B_k s}, \tag{4.9}$$

with s and y defined as:

$$s := m^{(k+1)} - m^{(k)} \qquad \text{and} \qquad y := \nabla \tilde{J}(m^{(k+1)}) - \nabla \tilde{J}(m^{(k)}). \tag{4.10}$$

The convergence of certain quasi-Newton methods, including the BFGS method, is proven to be locally superlinear, which is slower compared to the quadratic convergence of the Newton method (Broyden et al., 1973).

Algorithm 4 does not require B_{k+1} directly to determine the quasi-Newton direction, but only its inverse. One solution to avoid the inversion of every B_{k+1} is to use the Sherman-Morrison-Woodbury formula: it states that the inverse of a rank-k correction of a matrix can be computed by performing a rank-k correction to the inverse of that matrix. The corresponding rank-two

update for the BFGS method B_{k+1}^{-1} is:

$$B_{k+1}^{-1} = \left(I - \frac{sy^*}{y^*s}\right) B_k^{-1} \left(I - \frac{ys^*}{y^*s}\right) + \frac{ss^*}{y^*s}. \tag{4.11}$$

An alternative of the inverse update formula (4.11) is to use the numerically more stable Cholesky decomposition of B_{k+1}. This is particularly important for poorly conditioned matrices, and can also be updated using a rank-two update (Seeger, 2008; Fletcher, 1980; Gill and Murray, 1972). The computational cost of all three update formulae is $\mathcal{O}(n^2)$ where n is the dimension of the optimisation variables, plus the cost for evaluating the functional gradient.

The main disadvantage of the BFGS method is that it requires the storage of the Hessian approximation B_{k+1} or B_{k+1}^{-1}. Since this is typically a dense matrix, the total number of values to be stored is $\mathcal{O}(n^2)$. This can constrain the dimensionality of the parameters space significantly: for example, with $n = 10^6$ the required storage for the Hessian approximation with double precision floats would exceed 7 TB.

The limited memory BFGS (L-BFGS) method (Liu and Nocedal, 1989) circumvents this problem by not storing B_{k+1}^{-1} explicitly but rather by keeping only the l most recent s and y values from equations (4.10), where l is specified by the user. The quasi-Newton direction step can then be evaluated by applying the BFGS inverse update recursively starting with an estimate for B_{k+1-l}^{-1}, typically with the identity matrix. This approach reduces the computational cost and the storage requirement for the quasi-Newton update to $\mathcal{O}(ln)$. Coming back to the example where $n = 10^6$, a typical value of $l = 10$ would reduce the storage requirement from 3.5 TB to less than 40 MB. However, these advantages come with the caveat that the control variable converges only R-linearly to the optimal control (Liu and Nocedal, 1989).

A further extension to L-BFGS allows box constraints of the form $a \leq m \leq b$ and is known as L-BFGS-B (Byrd et al., 1995). L-BFGS-B belongs to the class of feasible optimisation algorithms, i.e. the optimisation variables satisfy the box constraints at each optimisation iteration. An initial optimisation variable that satisfies the constraints is obtained by projecting the optimisation variable onto the feasible subset.

More general equality and inequality constraints can be handled with sequential quadratic programming described in the next section.

4.3.2. Sequential quadratic programming

The SQP method is considered to be one of the most efficient algorithms for general non-linear optimisation problems (Hock and Schittkowski, 1983). In contrast to quasi-Newton methods such as BFGS, the SQP method can handle general equality and inequality constraints and is therefore suitable to solve the general form of the reduced optimisation problem (4.3). This chapter can only give an outline of the main concepts of SQP. A more detailed introduction can be found in Boggs and Tolle (1995) or Nocedal and Wright (1999, Chapter 18).

The basic idea of SQP is to approximate the reduced optimisation problem (4.3) by a quadratic subproblem and use its solution to find a better next iterate. The definition of the quadratic subproblem is based on the Lagrange functional of the reduced optimisation problem (4.3):

$$\mathcal{L}(m, \mu, \xi) \coloneqq \tilde{J} + \mu^* h(m) + \xi^* g(m),$$

where μ and ξ are the Lagrange multipliers. Given the current iterate for the control variable $m^{(k)}$ and Lagrange multipliers $\mu^{(k)}$ and $\xi^{(k)}$, the quadratic subproblem solved by SQP is based on a linearisation of the constraints and is defined as (Nocedal and Wright, 1999, §18):

$$\min_d \ \nabla \tilde{J}(m^{(k)})d + \frac{1}{2} d^* \frac{\partial^2 \mathcal{L}(m^{(k)}, \mu^{(k)}, \xi^{(k)})}{\left(\partial m^{(k)}\right)^2} d \tag{4.12a}$$

$$\text{subject to } \nabla h(m^{(k)})^* d + h_i(m^{(k)}) = 0, \tag{4.12b}$$

$$\nabla g(u^{(k)})^* d + g_i(m^{(k)}) \leq 0, \tag{4.12c}$$

which yields the solution d and associated Lagrange multipliers d_μ and d_ξ. These values are then used to update the current iterate. An outline of this basic SQP algorithm is given in algorithm 5.

The SQP method can be viewed as a natural extension of the Newton method and many techniques can be reapplied. In particular, the Hessian of the Lagrangian in (4.12a) can be approximated with a quasi-Newton technique just as in the BFGS method.

ALGORITHM 5: Local SQP algorithm

Data: initial values for the iterates $m^{(0)}, \mu^{(0)}$ and $\xi^{(0)}$
Result: optimal value m^{opt}
$k \leftarrow 1$
repeat
  $k \leftarrow k + 1$
  determine d, d_μ and d_ξ by solving the quadratic subproblem (4.12)
  update

$$m^{(k)} \leftarrow m^{(k-1)} + d$$
$$\mu^{(k)} \leftarrow d_\mu$$
$$\xi^{(k)} \leftarrow d_\xi$$

until *stopping criteria*

In contrast to the L-BFGS-B algorithm from the previous section, SQP is an infeasible algorithm, i.e. the constraints are not exactly satisfied during the optimisation procedure. In particular, the initial optimisation variable $m^{(0)}$ does not have to satisfy the (in-)equality constraints of (4.1), which can be an important property as finding a feasible initial value can be difficult for complex constraints.

4.4. The optimisation framework

The framework presented here is implemented as an extension module of dolfin-adjoint (chapter 3). The source code and the presented examples are available at `http://dolfin-adjoint.org`.

4.4.1. User interface

The optimisation framework relies on the tape that is recorded by dolfin-adjoint with the operator overloading technique described in chapter 3. Thus, the user first runs the forward model to create the associated tape of the forward model.

At that point, the user can define the reduced functional $\tilde{J}$ with

```
1 j_tilde = ReducedFunctional(j, m)
```

where `j` is a `Functional` object that specifies the functional of interest, and `m` is a `Parameter` object which describes the optimisation variables, see

section 3.3.4. Optionally, a list of `Parameter` objects can be provided, for example to optimise for multiple initial conditions. The reduced functional object `j_tilde` can then be directly used to solve the associated minimisation problem $\min_m \tilde{J}(m)$ by calling:

```
m_opt = minimize(j_tilde)
```

or the maximisation problem $\max_m \tilde{J}(m)$ by calling:

```
m_opt = maximize(j_tilde)
```

Both these functions solve the optimisation problem with the default settings and return the optimised parameters when finished.

Additional arguments are used to set and configure the optimisation method and to define box and (in-)equality constraints. Currently, the presented framework supports most of the algorithms in the optimisation package of SciPy (Jones et al., 2001). For problems without box and (in-)equality constraints these are:

- the Nelder-Mead method (Nelder and Mead, 1965),
- a modification to the Powell's method (Powell, 1964),
- a non-linear conjugate gradient method (Nocedal and Wright, 1999, §5.2),
- the BFGS method (section 4.3.1),
- the Newton-CG method (Nocedal and Wright, 1999, §7.1),
- and simulated annealing (Laarhoven and Aarts, 1987).

The Nelder-Mead, Powell's and simulated annealing methods are gradient-free optimisation algorithms. For these, the adjoint model is not required and the optimisation algorithm uses libadjoint's tape only to replay the forward model. Consequently, they also do not require the forward model to be differentiable. For problems with box or (in-)equality constraints, the user has the choice between:

- SQP (section 4.3.2),
- and the gradient-free Constrained Optimization by Linear Approximation (COBYLA) method (Powell, 1994).

Finally, if only box-constraints are applied, the user can additionally choose from:

- the L-BFGS-B method (section 4.3.1),
- and a Newton-CG implementation that supports box constraints.

The user also has access to more advanced options, such as to automatically test each gradient computation with the Taylor remainder convergence test (2.14) during the optimisation procedure, or to execute user specific code after each gradient computation, for example to create convergence plots.

4.4.2. Implementation

Objects of the `ReducedFunctional` class (`j_tilde` in section 4.4.1) have two main functionalities: they can evaluate the functional of interest, and its gradient for a given parameter value. The functional evaluation is implemented using the replay functionality of libadjoint (section 2.5.6) to rerun the forward model, while simultaneously evaluating the functional of interest. However, a direct replay would simply reevaluate the forward model with the original parameter values. This is resolved by generating new callback functions (section 3.3.3) of the annotated operators that define the parameter values, and by replacing the original callbacks. The gradient computation for a given parameter value first replaces the callbacks as above, then replays the forward model to obtain forward solutions for that parameter choice, and finally uses dolfin-adjoint to compute the gradient with the adjoint approach.

The `minimize` and `maximize` routines implement the interface to the supported optimisation algorithms. These algorithms commonly require the implementation of functions for evaluating the functional and the functional gradient. The `minimize` and `maximize` routines generate these functions using the `ReducedFunctional` object and by performing the conversions from DOLFIN data structures, such at functions and constants, to generic array data types on which the optimisation algorithms typically operate.

Parallel execution is often crucial in PDE-constrained optimisation to achieve reasonable run times, since one optimisation run consists of many functional and gradient evaluations, each of which requires computationally

expensive PDE solves. While DOLFIN supports the parallel solution of the forward and adjoint models, the considered implementations of the optimisation algorithms are not designed for distributed execution. The `minimize` and `maximize` routines solve this problem by executing the PDE solves in parallel, but replicating the optimisation computation on each processor. For that, the `minimize` and `maximize` functions serialise the distributed data structures of the optimisation variables, the functional gradient and the constraints, as these are needed for the optimisation algorithm. All other variables, in particular the forward and adjoint solutions, remain parallelised and are not communicated. This approach works well for small-scale optimisation problems, where the communication time for gathering the data and the execution time spent on the optimisation algorithm is small compared to the runtime of the PDE solves. For large-scale optimisation problems one should consider interfacing a parallel optimisation algorithm, such as TAO (Munson et al., 2012) or OPT++ (Meza, 1994).

There are cases where optimisation purely based on the annotation is not desired. For example, a forward model with an adaptive time stepping scheme changes the time step according to certain conditions. This adaptivity is not reflected in the annotation, and hence the optimisation would always be performed with the time step choices from the initial run. For such cases, the optimisation module allows the user to manually specify the evaluation function that may then update the annotation. However, this approach has the disadvantage that the discretised PDE changes in every optimisation iteration. If the optimal solutions are sensitive to these changes, this can result in a reduced convergence of the optimisation algorithm.

4.5. Examples

This section demonstrates the application of the presented optimisation framework to solve two classical optimal control problems. The first example solves the optimal heating problem (4.4). The governing PDE in this case is the stationary heat equation. The second example replaces the stationary heat equation with a time-dependent, non-linear PDE. Although this problem adds significant complexity to the forward and adjoint PDEs, the required code changes are minimal.

Further applications of this framework can be found the remaining chapters of this thesis.

4.5.1. Distributed control of the heat equation

The first example solves the optimal control problem of the stationary heat equation (4.4). The problem possesses a unique optimal solution if the box constraints limits a and b are bounded (Tröltzsch, 2005, chapter 2.5.1). In the case that one of the box constraints is unbounded, the regularisation term must be strictly positive, i.e. $\alpha > 0$, to ensure uniqueness. In the following example, these conditions are satisfied by choosing $\alpha = 0$, $a = 0$ and $b = 0.5$.

Since the presented framework applies the discretise-then-optimise approach, the first step consists of the problem discretisation. For that, the heat equation is discretised using the finite element method. The two-dimensional domain of interest $\Omega := [-1, 1]^2$ was uniformly discretised using triangular elements. Then, the finite-dimensional function spaces are constructed with piecewise linear continuous finite elements for the PDE-solution u and the desired temperature profile u_d and discontinuous piecewise constant finite elements for the heat source m. The latter choice is motivated by the fact that the control can possess discontinuities for $\alpha = 0$. The following code example initialises the domain, the function spaces and all required functions in DOLFIN:

```
# Define the domain [-1, 1] x [-1, 1]
n = 200
mesh = Rectangle(-1, -1, 1, 1, n, n)
# Define the function spaces
V = FunctionSpace(mesh, "CG", degree=1)
W = FunctionSpace(mesh, "DG", degree=0)
# Define the functions
u = Function(V, name="Solution")
m = Function(W, name="Control")
v = TestFunction(V)
```

The weak form of the heat equation is obtained by multiplying the PDE with a test function $v \in V$, then integrating the result over the domain and finally integrating by parts. The resulting weak formulation is: find $u \in V$

such that:

$$\langle \nabla u, \nabla v \rangle_\Omega = \langle m, v \rangle_\Omega \quad \forall v \in V. \tag{4.13}$$

The associated DOLFIN code is:

```
# Solve the forward model to create the tape
F = (inner(grad(u), grad(v)) - m*v)*dx
bc = DirichletBC(V, 0.0, "on_boundary")
solve(F == 0, u, bc)
```

During this initial solve of the heat equation, libadjoint records the tape of the forward model. This tape is used by the optimisation framework to compute all future functional evaluations and gradient computations.

Next, the functional is defined. The desired temperature profile u_d is defined as:

$$u_d(x, y) = e^{-1/(1-x^2)-1/(1-y^2)},$$

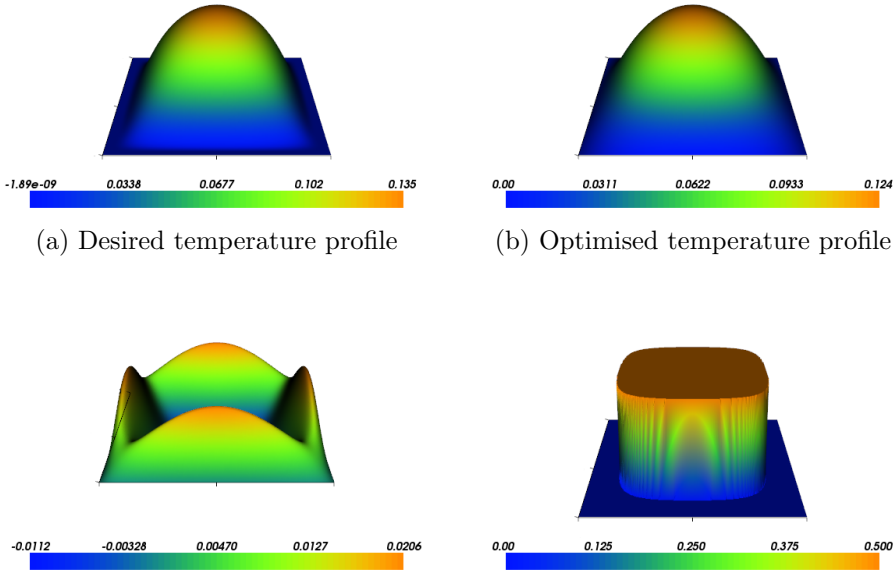
and plotted in figure 4.2a. With this, the functional of interest of the considered optimal control problem (4.4) can be defined using the corresponding dolfin-adjoint function described in section 3.3.4:

```
# Define the functional of interest
u_d = exp(-1/(1-x[0]*x[0])-1/(1-x[1]*x[1]))
j = Functional((0.5*inner(u-u_d, u-u_d))*dx*dt[FINISH_TIME])
```

Here, the `FINISH_TIME` expression indicates that the functional of interest is to be evaluated at the end of the computation. At this point, the optimisation framework has all required information to solve the optimal control problem:

```
# Define the reduced functional
j_tilde = ReducedFunctional(j, InitialConditionParameter(m))
# Solve the optimisation problem: min_m j_tilde(m)
m_opt = minimize(j_tilde, algorithm = "scipy.l_bfgs_b",
                 pgtol=2e-12, bounds = (0.0, 0.5))
```

The last two parameters of `minimize` are specific to the L-BFGS-B implementation: `pgtol` sets the termination condition to stop when the infinity norm of the projected gradient becomes less than $2 \cdot 10^{-12}$ and `bounds` specifies constant upper and lower values for the box constraints.



(a) Desired temperature profile

(b) Optimised temperature profile

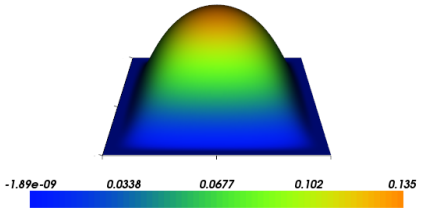
(c) Difference between optimised and desired temperature profiles

(d) Optimised heat source control

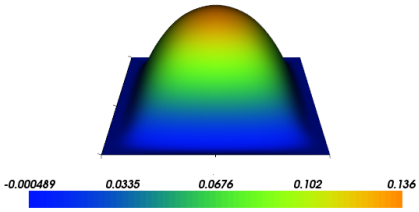
Figure 4.2.: The solutions of the stationary optimal heating problem with box constraints. The heat source control is limited by the box constraints in large parts of the domain, which leads to a relatively large difference between desired and optimised temperature profiles.

The optimisation algorithm starts with a zero estimate for the control, i.e. $m^{(0)} = 0$. The corresponding functional value is $\tilde{J}\left(m^{(0)}\right) = 8.9 \cdot 10^{-3}$. The L-BFGS-B algorithm terminates successfully after 22 iterations yielding a final functional value of $1.4 \cdot 10^{-4}$. The results are visualised in figure 4.2. The desired and optimised temperature profile are of similar shape, but their maximum values differ significantly. This is reflected in the fact that the box constraints are active in large parts of the domain (figure 4.3d).

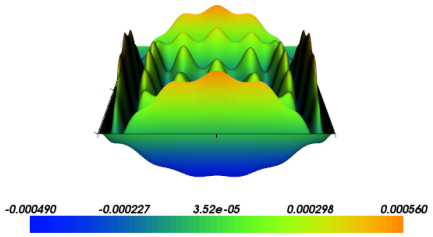
By removing the box constraints, a better agreement between the desired and optimised temperature profile is expected. Therefore the box constraints were removed and the regularisation parameter set to $\alpha = 10^{-7}$, in order to ensure the uniqueness of the optimal solution. The results are shown in figure 4.3. Compared to the previous solution, the difference between the optimised and the desired temperature profiles is decreased by



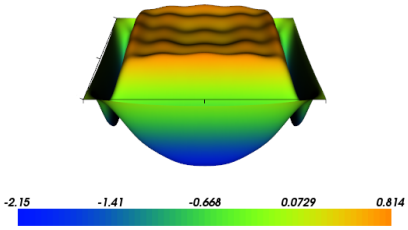
(a) Desired temperature profile



(b) Optimised temperature profile



(c) Difference between optimal and desired temperature profiles



(d) Optimised heat source control

Figure 4.3.: The solutions of the stationary optimal heating problem without box constraints. The optimised heat source control achieves a good agreement between the desired and optimised temperature profiles.

more than an order of magnitude. The final functional is $7.7 \cdot 10^{-8}$, and hence significantly smaller than with box constraints.

4.5.2. Distributed control of a non-linear, time-dependent PDE

This example extends the previous one by replacing the stationary heat equation with a time-dependent, non-linear PDE:

$$\min_m \frac{1}{2}\|u(t=T) - u_d\|^2_{L^2(\Omega)} + \frac{\alpha}{2}\|m\|^2_{L^2(\Omega)} \tag{4.14a}$$

$$\text{subject to } \frac{\partial u}{\partial t} - \nabla^2 u + u^3 = m \qquad \text{on } \Omega \times (0,T), \tag{4.14b}$$

$$u = 0 \qquad \text{on } \partial\Omega \times (0,T), \tag{4.14c}$$

$$u = 0 \qquad \text{for } \Omega \times \{0\}, \tag{4.14d}$$

$$a \leq m \leq b \qquad \text{on } \partial\Omega \times (0,T). \tag{4.14e}$$

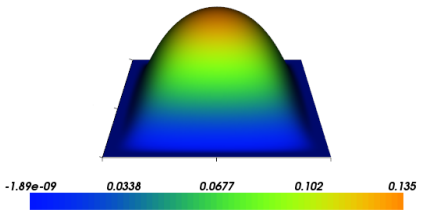
The state equation consists of the PDE with appropriate initial and boundary conditions. The functional measures the difference between the numerical and the desired temperature profiles at the final simulation time T.

The non-linearity and time-dependency of the state equation (4.14b)–(4.14d) adds significant complexity to the solution process of the optimisation algorithm. In particular, the gradient computations involve the storage of the whole forward solution (or the application of some checkpointing scheme) and the solution of the associated adjoint system. However, because the details of the gradient computation are abstracted away in the `compute_gradient` function of dolfin-adjoint, the optimisation module can apply exactly the same concepts to solve this problem.

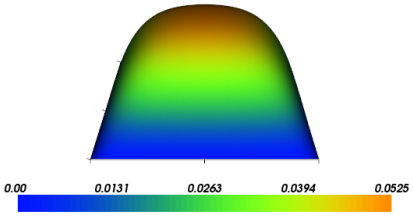
As a result, the implementation from the previous example can be reused almost entirely to solve problem (4.14). The only change required is to replace the weak formation of the heat equation with the new governing PDE. The following code shows an example implementation with a backward Euler time discretisation and a Newton solver for the non-linear equation solves:

```
# Define the weak form
timestep = 0.01
F = ((u - u_old)*v/timestep + inner(grad(u), grad(v)) + u**3*v
    - m*v)*dx
# Perform 10 timesteps
for t in range(11):
    solve(F == 0, u, bc)
    u_old.assign(u)
    adj_inc_timestep()
```

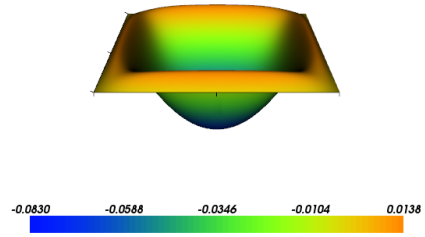
The optimisation algorithm starts with $m^{(0)} = 0$ and an initial functional value of $\tilde{J}\left(m^{(0)}\right) = 8.9 \cdot 10^{-3}$. The L-BFGS-B algorithm terminates successfully after 29 iterations with an optimised functional value of $2.5 \cdot 10^{-3}$. The results are visualised in figure 4.4. The control is equal to the upper bound



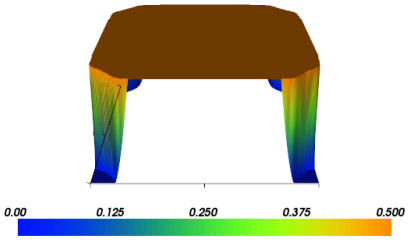
(a) Desired temperature profile



(b) Optimised temperature profile



(c) Difference between optimal and desired temperature profiles



(d) Optimised heat source control

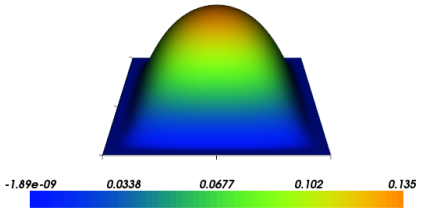
Figure 4.4.: The numerical solutions to the time-dependent optimal heating problem with box constraints. The heat source control is limited by the box constraint in most parts of the domain which leads to a large difference between the desired and optimised temperature profile.

in most of the domain, see figure 4.4d. This results in a large disagreement between the desired and the optimal temperature profiles, see figure 4.4c.

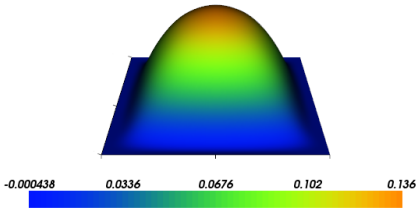
The results of the optimisation problem without box constraints and a regularisation coefficient of $\alpha = 10^{-7}$ is shown in figure 4.5. As expected, the difference between desired and optimised temperature profiles is significantly reduced and the optimised control values exceed the box constraints from the previous setup. The final functional value is $5.8 \cdot 10^{-7}$.

4.6. Verification

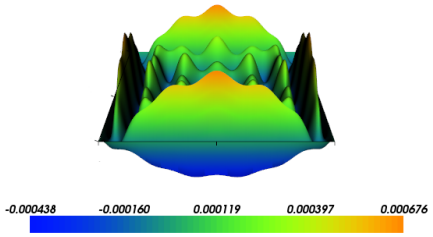
The optimisation framework is verified on problems with analytical solutions. With the analytical solution available, the numerical order of convergence can be determined experimentally and compared to the theory. The



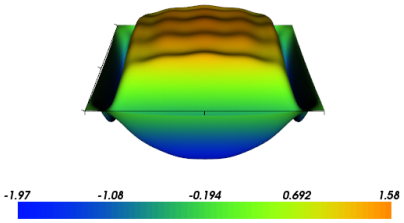
(a) Desired temperature profile



(b) Optimised temperature profile



(c) Difference between optimal and desired temperature profiles



(d) Optimised control

Figure 4.5.: The numerical solutions to the time-dependent optimal heating problem without box constraint. Without the box constraint the optimal control achieves a significantly better fit between desired and optimised temperature profiles

agreement of these two rates is considered to be a strong indicator that the implementation is correct (Salari and Knupp, 2000).

The considered analytical solutions are based on the optimal control problem (4.4) of the heat equation, extended with a source term s:

$$\begin{aligned} \min_{m} \ & \|u - u_d\|^2_{L^2(\Omega)} \\ \text{subject to } & -\nabla^2 u = m + s \quad \text{in } \Omega, \\ & u = 0 \quad \text{on } \partial\Omega, \\ & -1 \leq m \leq 1. \end{aligned} \tag{4.15}$$

To construct an analytical solution for this problem, first the optimal PDE solution u^{opt} and the optimal control parameter m^{opt} are chosen, such that they satisfy the initial and boundary conditions, and the box constraints.

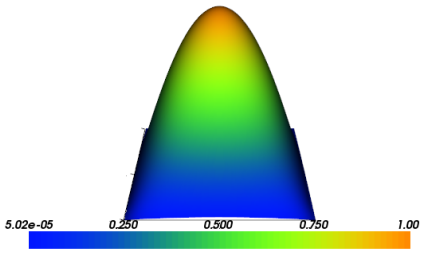


Figure 4.6.: The optimal control m^{opt} for the smooth control test.

Element size	$\|m - m^{\text{opt}}\|$	order	$\|u - u^{\text{opt}}\|$	order
$1.25 \cdot 10^{-1}$	$9.54 \cdot 10^{-2}$		$3.83 \cdot 10^{-4}$	
$6.25 \cdot 10^{-2}$	$3.70 \cdot 10^{-2}$	1.34	$9.03 \cdot 10^{-5}$	2.08
$3.12 \cdot 10^{-2}$	$1.71 \cdot 10^{-2}$	1.11	$2.20 \cdot 10^{-5}$	2.03
$1.56 \cdot 10^{-2}$	$8.33 \cdot 10^{-3}$	1.04	$5.43 \cdot 10^{-6}$	2.02
$7.81 \cdot 10^{-3}$	$4.11 \cdot 10^{-3}$	1.02	$1.36 \cdot 10^{-6}$	2.00

Table 4.1.: The rate of convergence for the smooth control test. The error in the control $\left\|m - m^{\text{opt}}\right\|$ is expected to decrease at first order, and the error in the PDE-solution $\left\|u - u^{\text{opt}}\right\|$ is expected to converge at second order.

With these, the source term and the desired PDE solution are defined as $s := -\nabla^2 u^{\text{opt}} - m^{\text{opt}}$ and $u_d := u^{\text{opt}}$. It is easy to see that m^{opt} and u^{opt} form an optimal solution to problem (4.15).

The following sections perform the two convergence tests, the first one with a continuous optimal control function, and the second one with a discontinuous optimal control function.

4.6.1. Smooth control

The first test defines a smooth optimal control function:

$$m^{\text{opt}}(x, y) := \sin(\pi x) \sin(\pi y) .$$

Figure 4.6 shows a plot of this optimal control m^{opt}. The optimal PDE solution is chosen to be:

$$u^{\text{opt}}(x, y) := \frac{1}{2\pi^2} \sin(\pi x) \sin(\pi y).$$

By substituting these solutions into the governing PDE of the optimal control problem (4.15), the source term evaluates to $s \coloneqq 0$. Finally, the desired PDE solution is $u_d \coloneqq u^{\text{opt}}$.

The discretisation and optimisation parameters are configured identically with the setup described in section 4.5.1, except a tighter termination tolerance for the optimisation algorithm of 10^{-16}. Then the convergence test was performed on five uniformly discretised meshes with decreasing mesh element sizes. The resulting errors and convergence rates are given in table 4.1. The first-order convergence for the control solution m is expected as the underlying function space is discretised with discontinuous, piecewise constant finite elements. Similarly, a second-order rate of convergence is observed for the PDE solution u, as it is discretised with continuous, piecewise linear finite elements.

4.6.2. Bang-bang control

The second verification test is motivated by the fact that box constraints can lead to optimal control solutions with discontinuities. The following example, derived in Tröltzsch (2005, chapter 2.9.1), is constructed such that the control variable yields discontinuities in a chessboard like shape:

$$m^{\text{opt}}(x, y) \coloneqq -\text{sign}\left(-\sin\left(8\pi x\right)\sin\left(8\pi y\right)\right).$$

A plot of the resulting optimal control is given in figure 4.7. This kind of control, where the control values jump from one box constraint limit to the other, is also known as bang-bang control. The optimal state solution is chosen to be:

$$u^{\text{opt}}(x, y) \coloneqq \sin(\pi x)\sin(\pi y).$$

By applying the optimality conditions, the source term s and the desired PDE solution u_d are obtained (Tröltzsch (2005, chapter 2.9.1)):

$$s(x, y) \coloneqq 2\pi^2 \sin(\pi x)\sin(\pi y) + \text{sign}\left(-\sin\left(8\pi x\right)\sin\left(8\pi y\right)\right),$$

and:

$$u_d(x, y) \coloneqq \sin(\pi x)\sin(\pi y) + \text{sign}\left(-\sin\left(8\pi x\right)\sin\left(8\pi y\right)\right).$$

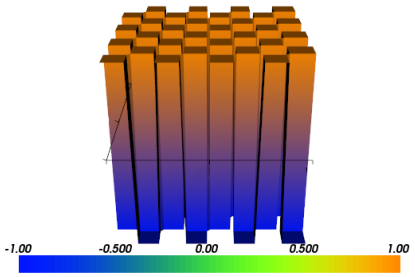


Figure 4.7.: The optimal control m^{opt} for the bang-bang test.

Element size	$\|m - m^{\text{opt}}\|$	order	$\|u - u^{\text{opt}}\|$	order
$3.13 \cdot 10^{-2}$	$4.76 \cdot 10^{-2}$		$7.38 \cdot 10^{-4}$	
$1.56 \cdot 10^{-2}$	$2.41 \cdot 10^{-2}$	0.99	$2.01 \cdot 10^{-4}$	1.89
$7.81 \cdot 10^{-3}$	$1.20 \cdot 10^{-2}$	1.00	$5.04 \cdot 10^{-5}$	1.99
$3.91 \cdot 10^{-3}$	$5.30 \cdot 10^{-3}$	1.18	$1.24 \cdot 10^{-5}$	2.03
$1.95 \cdot 10^{-3}$	$2.30 \cdot 10^{-3}$	1.20	$2.54 \cdot 10^{-6}$	2.29

Table 4.2.: The rate of convergence for the bang bang control test. The error in the control $\left\|m - m^{\text{opt}}\right\|$ is expected to decrease at first order, and the error in the PDE-solution $\left\|u - u^{\text{opt}}\right\|$ is expected to converge at second order.

The convergence test is performed with the same configuration as in the previous test. The resulting errors and convergence rates are given in table 4.2. It shows the expected first-order convergence for the control and second-order convergence for the state solution.

4.7. Summary

A new framework for rapidly defining and solving PDE-constrained optimisation problems was developed. It uses the FEniCS framework, libadjoint and dolfin-adjoint to achieve a high-level of automation. This automation is the result of performing all required tasks for the optimisation purely on the tape that is recorded by libadjoint. This includes running the forward model, evaluating the functional of interest, deriving and solving the adjoint problem to compute the functional gradient, and modifying the tape in order to incorporate updated parameter values.

By interfacing with external optimisation packages, the user has the choice among a variety of gradient-free and gradient-based methods. General (in-)equality and box control constraints can be applied, if supported by the selected optimisation algorithm. As demonstrated, the optimisation framework works with both linear and non-linear as well as both steady-state and time-dependent governing PDEs.

The following two chapters apply this optimisation framework to two novel problems.

Chapter 5

Layout optimisation of marine turbine arrays

Contents

5.1. Introduction 119

5.2. Current methods in wind farm layout optimisation 121

5.3. Mathematical formulation as a PDE-constrained optimisation problem 123

5.4. Numerical setup 126

5.5. Verification 130

5.6. Examples 132

5.7. Conclusions 140

Abstract

Tidal turbine arrays are considered as one of the most promising technologies for extracting energy from tidal streams. In order to generate a signification amount of power, up to hundreds of tidal turbines must be deployed. The positioning of these turbines in the flow can significantly influence the extracted power, and hence is of huge economic interest. In this chapter, this layout problem is formulated as an optimisation problem and solved using efficient gradient-based optimisation algorithms. In each optimisation iteration, a shallow water model predicts the flow and the performance of the current turbine layout.

The capabilities of this approach is demonstrated by optimising turbine locations in three idealised scenarios, which increases the power extraction between 20% to 75% compared to a reference layout.



Figure 5.1.: Offshore wind farm in the North Sea at Horns Rev, Denmark. The 80 turbines are laid uniformly in the shape of an oblique rectangle. What is the layout for marine turbines that maximises the power outcome?
Image copyright: Elsam A/S

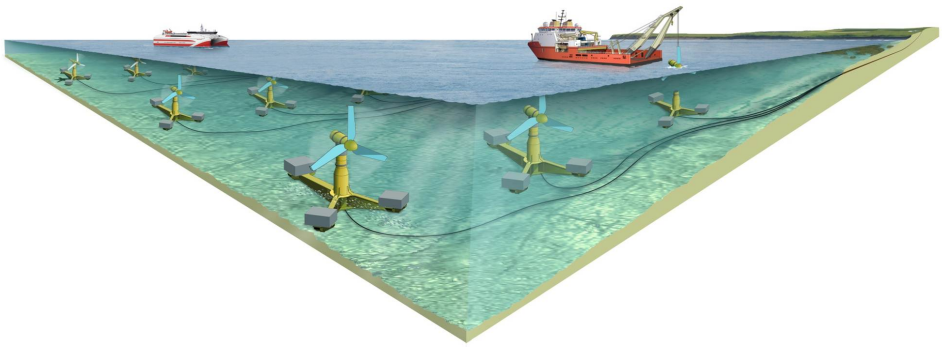


Figure 5.2.: Artistic image of the deployment of tidal stream turbines. The positions of the turbines have a significant influence on the extracted power output and is hence economically crucial.
With kind permission of MeyGen Ltd. Image copyright: MeyGen Ltd.

5.1. Introduction

With the increasing cost of energy, tidal turbines are becoming a competitive and promising option for renewable electricity generation. In order to extract a significant amount of energy, arrays consisting of hundreds of tidal turbines must be deployed in a tidal stream. This raises the question of where to place the turbines in order to maximise the power output; finding the optimal configuration is of huge importance as it can substantially change the energy captured and determine whether the project is economically viable. However, the determination of the optimal layout is difficult because of the complex flow interactions between turbines and the fact that the power output depends sensitively on the flow velocity at the turbine position.

This problem has heretofore been addressed in two different ways. One approach is to simplify the tidal flow model such that the solutions are either available as explicit analytical expressions for which the optimum can then be derived directly, or the solution is sufficiently cheap such that the whole parameter space can be explored. For example, Bryden and Couch (2007) and Garrett and Cummins (2008) optimised simplified models to derive an estimate for the maximum energy that can be extracted from a tidal basin. Vennell (2010, 2011) used simple one-dimensional models to investigate the influence of different farm parameters such as farm's drag coefficient and the number of turbine rows to deduce optimal configurations. While this approach can provide a coarse estimate for the power potential of a site, these simplified equations do not accurately capture the complex non-linear flow interaction between turbines.

The second approach is to use more realistic PDE-based models to accurately predict the tidal flow and the resulting power output. For these models, their computational expense prohibits the exploration of the whole parameter space. Consequently, usually only some selected turbine layout configurations are investigated. Divett et al. (2011) compared the power output of four different configurations by solving the two-dimensional non-linear shallow water model and was able to improve the power outcome by over 50% compared to a regular layout. Lee et al. (2010) used a three-dimensional model to investigated how the distance between adjacent rows in a regular array layout impacts the turbine efficiency and showed an ef-

ficiency decay for distances of less than three times the turbine diameter. While these studies show the potential of improving the performance by changing the turbine positions, such manual optimisation guided by intuition and experience becomes difficult in a realistic domain with complex bottom bathymetry, flow dynamics and hundreds of turbines.

In this chapter, the limits of these two approaches are overcome by applying a gradient-based optimisation algorithm to find optimal turbine configurations using a two-dimensional PDE based model. Because of the computational expense of this model, it is crucial to choose optimisation algorithms that strive to minimize the number of model evaluations. Furthermore, in order to be able to optimise hundreds of turbines, it is essential to ensure that the computational cost of each optimisation iteration scales well with the number of parameters and therefore the number of turbines.

Optimisation algorithms can be classified into two categories: gradient-free and gradient-based algorithms. Gradient-free optimisation algorithms use the functional of interest as a black box. They proceed by evaluating the functional at many points in parameter space and use these values to decide which areas of parameters space merit further exploration. While these methods tend to be robust and can, under certain smoothness conditions, provably find globally optimal solutions (Rudolph, 1996), they typically require a large number of functional evaluations that scales linearly or superlinearly with the number of parameters to be optimised. For example, in Bilbao and Alba (2009) used a genetic algorithm, that mimics the process of natural evolution, to optimise the location of 8 wind turbines. The algorithm was able to improve the power output by about 70% compared to the initial layout after $17,300$ functional evaluations. This large number of evaluations clearly introduces a practical upper limit for the number of turbines that can be optimised. This difficulty is compounded if a more realistic, and hence costly PDE based model is used. This issue is generally overcome by using simplified models where the functional of interest is a closed form function of the turbine position to achieve fast evaluation run times.

Gradient-based optimisation algorithms use additional information to update the position in parameter space at each iteration: the derivative of the functional of interest with respect to the parameters. Depending on the problem, this can lead to a significant reduction in the number of iterations

compared to gradient-free algorithms, making it suitable for large scale optimisation problems (Biegler et al., 2003). One caveat of applying gradient-based optimisation algorithms is that they find only local optima. This issue can be circumvented by hybrid approaches (Huang, 2009). Another caveat is that the determination of gradient information can be difficult for complex models.

In this chapter, the gradient information is computed with the adjoint approach as described in chapters 2 and 3. While the adjoint method is well established in certain scientific areas such as meteorology (Errico, 1997) and ocean modelling (Menemenlis et al., 2008), it is not well known for the layout problem at hand. As far as the author is aware, this chapter presents the first application of the adjoint method to optimise the marine turbine positions based on a non-analytic flow model. While the examples are shown in marine environment, it is expected that the presented techniques can be reapplied to the layout optimisation of wind farms.

5.2. Current methods in wind farm layout optimisation and possible adaptations to the marine environment

Layout optimisation for wind farms has been focus of numerous studies, most of which are based on gradient-free optimisation algorithms. In particular, evolutionary methods (Bäck, 1996) are known to yield good results (Salcedo-Sanz et al. (2011) and the references therein). These algorithms mimic the process of natural evolution by considering a population of candidate solutions on which it executes an evolutionary process to find the "fittest" solution. An related method is the particle swarm optimisation (Kennedy and Eberhart, 1995), which considers a population of candidate solutions called particles, that move through the parameter space and influence each other to drive the swarm to the best solution. Wan et al. (2010) applied particle swarm optimization on an analytical wake model to optimise up to 39 turbines with an analytical wake model and showed that this approach can yield better optimal solutions than genetic algorithms. Simulated annealing algorithms (Laarhoven and Aarts, 1987) are probabilistic optimisation algorithms that exploit an analogy between the

way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system. Bilbao and Alba (2009) used an analytical wake model to compare the simulated annealing algorithm with genetic optimisation algorithm. In a scenario with 47 turbines, the number of model evaluations could be reduced from $1,036,200$ with the genetic algorithm to $61,802$ with simulated annealing.

Only few publications solve the layout problem with gradient-based optimisation algorithms. Lackner and Elkinton (2007) applied gradient-based optimisation algorithms for which the energy production model was simplified such that the derivative information is available analytically. Huang (2009) combined a genetic algorithm with a hill climbing method to achieve a faster convergence to an optimal solution. With this additional information, the number of iterations were reduced by approximately an order of magnitude, with a similar optimal functional output. Finally, Fagerfjäll (2010) showed how mixed integer linear programming techniques can be used to optimise for both the number and position of turbines. However, the mentioned publications use very simplified wake models for which the gradient computation is either available analytically or can be easily approximated.

In realistic domains, such simple models are unlikely to simulate the complex flow around the turbines with sufficient accuracy. There has been much effort into simulating wind turbines with advanced numerical techniques, such as three-dimensional Navier-Stokes simulations and different turbine parametrisations (Mikkelsen, 2003; Pape and Lecanu, 2004). Can these techniques be adapted to the marine situation considered here? Turbine parametrisations such as the actuator disk method and the increased roughness coefficient parameterisation (Vermeer et al., 2003) have been applied in the marine environment (Ben Elghali et al., 2007; Divett et al., 2011). However, there are some key differences between the flow physics in the marine and atmospheric environment. Firstly, the flow in a tidal channel is dominantly driven by the predictable tidal forcing, while wind flow modelling needs to include the temporal uncertainty of magnitude and direction of the wind forcing. Secondly, the ratio of turbine height and free surface elevation is significantly different: while the rotor diameter of a wind turbine is small compared to the height of the atmosphere, tidal turbines

have typically diameters of about 20 m and are deployed in water depths of about 50 m or less. This leads to little undisturbed flow above the turbine which potentially increases the length of the wake compared to wind turbines (Divett et al., 2011).

5.3. Mathematical formulation as a PDE-constrained optimisation problem

In this section, the optimal positioning of turbines is formulated as a PDE-constrained optimisation problem in the generic form (4.1). The functional of interest J computes the extracted of the marine turbines. The flow is computed by the shallow water equations. The optimisation variable m encodes the x and y positions of N turbines in the format:

$$m = \boxed{p_1^x} \boxed{p_1^y} \boxed{p_2^x} \boxed{p_2^y} \boxed{\dots} \boxed{p_N^x} \boxed{p_N^y}$$

Box constraints of the form (4.2) are used to restrict the valid turbine positions to a rectangular site. Finally, the inequality constraints are used to enforce additional conditions on the turbine layout, for example a minimum distance between any two turbines.

5.3.1. The governing PDE

The turbine layout problem is here constrained by the non-linear shallow water equations with appropriate initial and boundary conditions:

$$\begin{aligned} \kappa\frac{\partial u}{\partial t} + u\cdot\nabla u - \nu\nabla^2 u + g\nabla\eta = \frac{c_b + c_t(m)}{H}\|u\|u \quad & \text{on } \Omega\times(0,T), \\ \kappa\frac{\partial \eta}{\partial t} + \nabla\cdot(Hu) = 0 \quad & \text{on } \Omega\times(0,T), \end{aligned} \tag{5.1}$$

where the unknowns u and η are the depth-averaged velocity and the free-surface displacement, respectively, H is the water depth at rest, g is the gravitational force, ν is the viscosity coefficient, and c_b and c_t are the coefficients for the quadratic bottom friction and the turbine parameterisation, respectively. The parameter $\kappa \in \{0, 1\}$ specifies if the stationary ($\kappa = 0$) or the non-stationary problem ($\kappa = 1$) is considered. The boundary conditions are as follows: on the domain inflow boundary Ω_{in} the velocity value

is enforced with a Dirichlet boundary condition. On the outflow boundary Ω_{out} the free-surface displacement is set to zero. Elsewhere, a no-normal flow boundary condition is imposed.

5.3.2. The turbine parameterisation

A turbine is modelled as increased friction in the concerned area (denoted as c_t in equation (5.1)). This approach is also used in Divett et al. (2011) where the authors set c_t to a constant value at the turbine locations and zero everywhere else. However, this parameterisation is problematic in the context of gradient-based optimisation because the friction becomes a non-differentiable function of the turbine position. For this reason, the turbine parameterisation used here smoothly increases the friction value at each turbine position. The associated friction function is constructed from bump functions, i.e. smooth functions with a finite support. A bump function in one dimension is:

$$\psi_{p,r}(x) \coloneqq \begin{cases} e^{1-1/(1-\|\frac{x-p}{r}\|^2)} & \text{for } \|\frac{x-p}{r}\| < 1, \\ 0 & \text{otherwise,} \end{cases} \tag{5.2}$$

where the two parameters p and r are the center and the support radius of the bump, respectively. A two-dimensional bump function is obtained by multiplying equation (5.2) in both independent dimensions. With that, the friction function of a single turbine is defined as:

$$C_{p,r}(x, y) \coloneqq K\psi_{p,r}(x)\psi_{p,r}(y), \tag{5.3}$$

where K is an additional scaling factor. A plot of the resulting friction for $K = 1$, $p = (10, 10)$ and $r = 10$ m is shown in figure 5.3. The turbine friction function c_t in the governing PDE (5.1) is defined to be the sum of the friction functions (5.3) for all N turbines:

$$c_t \coloneqq \sum_{i=1}^{N} C_{p_i,r}, \tag{5.4}$$

where a single value for r is used based on the assumption that the deployed turbines are of equal size. Note that additional turbine properties can be realised by using a generalised scaling parameter K in equation (5.3).

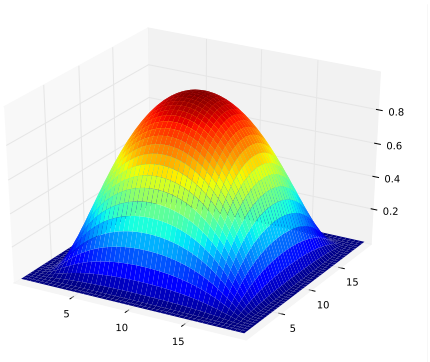


Figure 5.3.: The turbine is parametrised by a smoothly increasing friction coefficient towards the turbine center, given by the bump function (5.2) multiplied in the x and y-direction.

For example, the amount of energy that is extracted from the individual turbines (e.g. due to different pitch settings of the turbine blades) can be controlled by defining different scaling factors K for each turbine. As demonstrated later, these factors can then be considered as part of the optimisation problem. Other possibilities are to fit the turbine parameterisation to measurements or to handle cut in/out velocities in which the turbines are operational. These extensions are not considered in this work.

5.3.3. The functional of interest

A natural choice for the functional of interest is the total extracted energy due to the increased friction in the tidal farm (Vennell, 2012). In the non-stationary case ($\kappa = 1$) this is expressed as:

$$J(u, m) = \int_0^* \int_\Omega \rho c_t(m) \|u\|^3 \mathrm{d}\Omega \, \mathrm{d}t, \tag{5.5}$$

where ρ is the density of the fluid. In the stationary case ($\kappa = 0$) the functional is defined to be the extracted power from the increased friction:

$$J(u, m) = \int_\Omega \rho c_t(m) \|u\|^3 \mathrm{d}\Omega. \tag{5.6}$$

More advanced functional choices could take into account the economical cost of the turbine farm, such as the installation and service costs depending

on turbine size and the deployment location, as well as potential environmental impacts. These are not considered in this study.

5.3.4. Box and inequality constraints

The box and inequality constraints in the generic optimisation problem (4.1) are used to define the feasible values for the optimisation variables. In the context of the turbine layout problem, a typical condition is to restrict the area in which the turbines may be placed to the development site. The numerical examples in this work have rectangular shaped deployment sites and therefore box constraints are sufficient to enforce this restriction. For more general site shapes appropriate inequality constraints would need to be used instead.

Another common condition is to ensure that individual turbines do not overlap. This is implemented by enforcing a minimum distance $d_{\min}$ between any two turbines:

$$\|p_i - p_j\|^2 \geq d_{\min}^2 \qquad \forall\ i, j : 1 \leq i < j \leq N.$$

More advanced constraints could enforce minimum or maximum deployment depth or to limit the maximum local bathymetry steepness where turbines may be installed, but are not investigated in this work.

5.4. Numerical setup

5.4.1. Optimisation algorithm

The turbine layout problem is solved in the reduced form (4.3) with the techniques described in chapter 4. A suitable optimisation algorithm must support both the box constraints and the inequality constraints discussed in the previous section. From the algorithms presented in chapter 4 only sequential quadratic programming (SQP) fulfills these requirements and is therefore used for the following numerical examples. Note that the SQP method, as most gradient-based optimisation algorithms, does not guarantee to find a global optimum. This downside can be overcome by combining global and local algorithms (Powell et al., 1991; Renders and Flasse, 1996) but is not considered in this work.

The SQP implementation used here is part of the SciPy optimisation package (Jones et al., 2001) and is described in Kraft (1988). This implementation is not scale-invariant, that is scaling the functional of interest can affect the convergence of the algorithm. Preliminary numerical investigation found that such rescaling was necessary for achieving fast convergence. For the numerical results presented here, the functional was scaled such that the maximum value of the initial gradient was ten times the turbine radius.

A simple performance comparison was undertaken by testing SQP against the limited memory version of the Broyden-Fletcher-Goldfarb-Shanno method with bound support (L-BFGS-B), as described in section 4.3.1. Because L-BFGS-B does not support inequality constraints, the test did not include the minimum distance constraints. The application of both optimisation algorithms on several small layout problems showed similar performance, with slightly faster convergence results with SQP.

5.4.2. Spatial discretisation

The governing PDEs (5.1) are discretised with the finite element method. The weak form is derived by multiplying the equations with test functions (Ψ, Φ) on suitable function spaces, integrating the result over the computational domain and integrating the appropriate terms by parts. Denoting the L^2 inner product as $\langle a, b\rangle_\Omega \coloneqq \int_\Omega a \cdot b \ \mathrm{d}x$, the weak form of equations (5.1) is: find (u, η) such that $\forall\ (\Psi, \Phi)$:

$$\begin{aligned} \kappa \left\langle \frac{\partial u}{\partial t}, \Psi \right\rangle_\Omega + \langle u \cdot \nabla u, \Psi\rangle_\Omega + \nu \langle \nabla u, \nabla \Psi\rangle_\Omega + g \langle \nabla \eta, \Psi\rangle_\Omega = \\ \left\langle \frac{c_b + c_t(m)}{H} \|u\| u, \Psi \right\rangle_\Omega, \\ \kappa \left\langle \frac{\partial \eta}{\partial t}, \Phi \right\rangle_\Omega - \langle Hu, \nabla \Phi\rangle_\Omega + \langle Hu \cdot n, \Phi\rangle_{\partial\Omega_{\mathrm{in}} \cup \partial\Omega_{\mathrm{out}}} = 0. \end{aligned} \tag{5.7}$$

The no-normal flow condition on $\partial\Omega \setminus (\partial\Omega_{\mathrm{in}} \cup \partial\Omega_{\mathrm{out}})$ is weakly imposed by excluding the associated surface integral in the continuity equation. The Dirichlet boundary conditions are strongly imposed by restricting the function spaces to functions that yield the correct boundary values.

The discretised problem is obtained by choosing discrete function spaces in the weak formulation (5.7). In this work, these are constructed from a suitable triangulation of the computational domain Ω using the Taylor-

Hood finite element pair which uses piecewise quadratic functions on each triangle for the velocity and piecewise linear functions on each triangle for the free-surface displacement (Taylor and Hood, 1973).

5.4.3. Temporal discretisation

If the non-stationary problem is considered, that is if $\kappa = 1$, then the spatially discretised equations (5.7) have to be discretised in time before they can be solved numerically. In the context of this work, the time-discretisation is performed using the implicit Euler method. For brevity, the equations (5.7) are written in the shortened form:

$$\begin{aligned} \left\langle \frac{\partial u}{\partial t}, \Psi \right\rangle_\Omega &= S_u(u, \eta), \\ \left\langle \frac{\partial \eta}{\partial t}, \Phi \right\rangle_\Omega &= S_\eta(u, \eta). \end{aligned}$$

Let the superscript n denote the time level. The implicit Euler method computes the unknowns for time level $n+1$ using a time step Δt by solving the non-linear system:

$$\begin{aligned} \left\langle \frac{u^{n+1} - u^n}{\Delta t}, \Psi \right\rangle_\Omega &= S_u(u^{n+1}, \eta^{n+1}), \\ \left\langle \frac{\eta^{n+1} - \eta^n}{\Delta t}, \Phi \right\rangle_\Omega &= S_\eta(u^{n+1}, \eta^{n+1}). \end{aligned} \tag{5.9}$$

Since all operators featuring in equations (5.9) (or equations (5.7) in the stationary case) are continuously differentiable, Newton's method can be applied for solving the non-linear systems. In this work, the Newton iteration was configured to compute the solution up to a residual tolerance of 10^{-10}.

5.4.4. Gradient computation

The functional gradient evaluations that are needed for the optimisation algorithm are computed with the adjoint approach (Gunzburger, 2003; Giles and Pierce, 2000). The first step of the gradient computation consists of

solving the adjoint equation for the adjoint solution λ:

$$\frac{\partial F^*}{\partial z}\,\lambda = \frac{\partial J}{\partial z},$$

where $z \coloneqq (u, \eta)^*$. These linear PDEs are derived from the governing PDEs (5.1) and the functional of interest from section 5.3.3 and is in this case:

$$\begin{aligned} -\kappa\frac{\partial \lambda_u}{\partial t} + (\nabla u)^*\lambda_u - (\nabla\cdot u)\,\lambda_u - u\cdot\nabla\lambda_u - \nu\nabla^2\lambda_u \\ -H\nabla\lambda_\eta - \frac{c_b + c_t(m)}{H}\left(\|u\|\lambda_u + \frac{u\cdot\lambda_u}{\|u\|}u\right) = \frac{\partial J^*}{\partial u}\,, \\ -\kappa\frac{\partial \lambda_\eta}{\partial t} - g\nabla\cdot\lambda_u = 0. \end{aligned} \tag{5.10}$$

where $\lambda \coloneqq (\lambda_u, \lambda_\eta)$ is the vector containing the unknown adjoint velocity and adjoint free-surface displacement, respectively. The derivation of these equations can be found in appendix C. The non-stationary adjoint equations have a initial-condition that sets the adjoint velocity and free-surface displacement at the final time to zero. Equations (5.10) are therefore solved backwards in time. The boundary conditions are the homogenised versions of the boundary conditions given in the governing PDEs (5.1).

Solving the continuous adjoint equations (5.10) to obtain the gradient information yields a numerical approximation of the continuous gradient. However, this gradient does in general not correspond exactly with the gradient of the discretised forward model (Gunzburger, 2003, §4.1.2). In the context of optimisation, this inconsistency can degrade or even stop the convergence of the optimisation algorithm (Griesse and Walther, 2004; Gunzburger, 2003, §4.1.2). Therefore, in this work the adjoint equations are derived from the discretised forward system. This approach has two main advantages. Firstly, using the techniques from chapters 2 and 3 the derivation and solution of the discrete adjoint equations are automated. Secondly, the resulting gradient is consistent with the discrete forward model.

The second step of the gradient computation consists of evaluating the functional gradient:

$$\frac{dJ}{dm} = -\lambda^*\frac{\partial F}{\partial m} + \frac{\partial J}{\partial m}.$$

Deriving the terms for the problem considered here yields:

$$\frac{dJ}{dm} = \left\langle \lambda_u^*, \frac{\partial c_t(m)}{\partial m} u \|u\| H^{-1} \right\rangle_{\Omega\times(0,T)}.$$

As the adjoint solution, the gradient evaluation is automated by the techniques presented in chapters 2 and 3.

Note that the main computational expense in the gradient computation is the solution of the adjoint equation (5.10), consisting of solving one linear PDE, independent of the choice of optimisation variables. In particular, the gradient computation is essentially independent of the number of turbines that are to be optimised, which is a crucial feature if many turbines are to be considered.

5.5. Verification

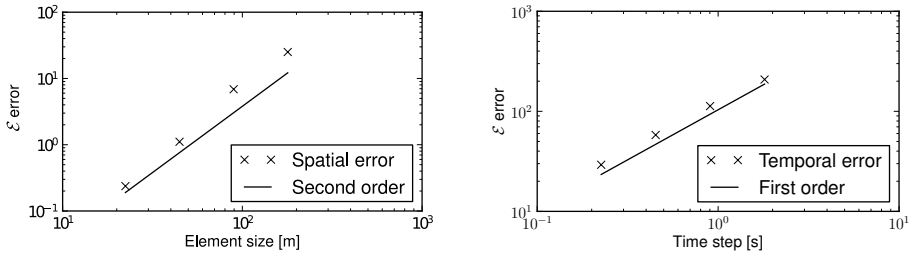
5.5.1. Verification of the forward model

The shallow water model implementation was verified by a convergence analysis based on an analytical solution. This analytical solution is constructed using the method of manufactured solutions (Salari and Knupp, 2000; Roache, 2002): A desired analytical solution is chosen and then substituted into the governing PDE, which yields a non-zero remainder. By adding this remainder as a source term to the governing equations, the selected solution becomes an analytical solution to the modified PDE.

For the following tests, the analytical solution consists of sinusoidal functions for both the velocity and the free-surface displacement:

$$\begin{aligned} u_{\text{exact}}(x,y,t) &= \begin{pmatrix} \eta_0 \sqrt{gH^{-1}} \cos\left(kx - \sqrt{gH}kt\right) \\ 0 \end{pmatrix}, \\ \eta_{\text{exact}}(x,y,t) &= \eta_0 \cos\left(kx - \sqrt{gH}kt\right), \end{aligned} \tag{5.11}$$

with $k = \pi/640$ m, $\eta_0 = 2$ m, $H = 50$ m, $\nu = 3\ \text{m}^2\text{s}^{-1}$, $c_t = 0$, $c_b = 0.0025$, $g = 9.81\ \text{ms}^{-2}$ and a final time of $T = \pi/(\sqrt{gH}k) \approx 28.9$ s which corresponds to half a wave cycle. The computational domain Ω is defined to be a rectangle of size 640 m $\times$ 320 m. Substituting the functions (5.11) into the shallow water equations (5.1) yields a non-zero remainder which is



(a) The spatial order of convergence (b) The temporal order of convergence

Figure 5.4.: The expected and achieved orders of convergence for the forward model.

added as a source term, and makes u_{exact} and η_{exact} analytical solutions to this modified PDE.

To determine the spatial order of convergence, the time step was fixed to a small value of $\Delta t = T/16,800$ s, to ensure that the numerical error of the spatial discretisation dominates the overall discretisation error. Then, the forward model with the added source term was run on four uniform, increasingly fine meshes and the error in the numerical solution (u, η) measured as:

$$\mathcal{E} = \left(\langle u - u_{\text{exact}}, u - u_{\text{exact}} \rangle_{\Omega \times (0,T)} + \langle \eta - \eta_{\text{exact}}, \eta - \eta_{\text{exact}} \rangle_{\Omega \times (0,T)} \right)^{\frac{1}{2}}.$$

The resulting errors plotted in figure 5.4a show the second-order convergence that is expected from the Taylor-Hood finite-element pair.

For determining the temporal order of convergence, a mesh with 2.5 m element size in the x-direction and 160 m element size in the y-direction was generated (the analytical solution does not vary in the y-direction and hence a relatively large mesh element size can be used). This mesh resolution ensures that the numerical error of the temporal discretisation dominates the overall discretisation error. The forward model was then run with a set of different time steps. The resulting errors plotted in figure 5.4b shows the first-order convergence expected for the implicit Euler time discretisation.

5.5.2. Verification of the gradient computation

The adjoint model and gradient computation was verified with the Taylor remainder convergence test described in section 2.5.6.

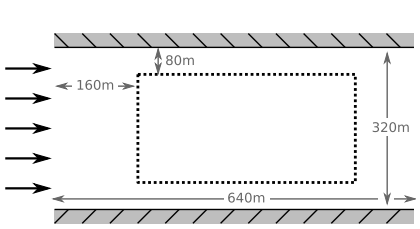
The Taylor remainder convergence test was applied in two stages. First a simple configuration with a single turbine was set up with the turbine parameterisation and the functional of interest as described above. The Taylor remainder convergence test in a random directions yielded the expected second-order convergence.

In addition, the Taylor remainder convergence test were applied on the numerical examples presented in the following section. For that, the setups were altered such that every gradient computation in the optimisation iterations was automatically checked with the Taylor remainder convergence test. To reduce the computational expense of these tests, the optimisation algorithm was aborted after five iterations. All the tests resulted in an order of convergence above 1.9, giving confidence that the adjoint model and the gradient computation are implemented correctly.

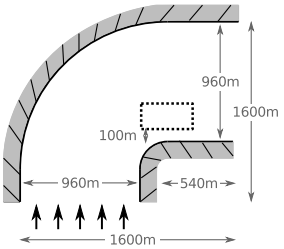
5.6. Examples

The following numerical examples solve the optimal layout problem in three idealised scenarios, shown in figure 5.5. In all examples, 32 turbines are to be deployed in a rectangular turbine site of size 320 m × 160 m. The idealised domains simplify the subsequent interpretation of the optimised turbine layout. Nevertheless, the domains are chosen such that they resemble structures that can be found in practical deployment sites. The main three objectives for the numerical examples are to investigate: By how much can layout optimisation increase the energy extraction? Can the optimisation algorithm reliably improve the energy extraction for different scenarios? How does the choice of the optimisation variables and the constraints impact the resulting layout?

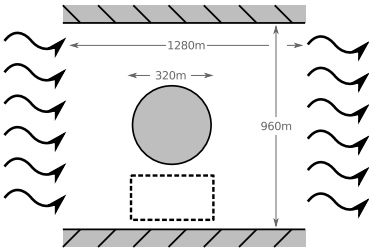
The simulations are based on the following parameters. A water depth of $H = 50$ m, a viscosity coefficient of $\nu = 3\ \mathrm{m^2 s^{-1}}$, and a gravitational force of $g = 9.81\ \mathrm{ms^{-2}}$ is used throughout. The bottom friction coefficient is set to $c_b = 0.0025$ which is a common value for coastal modelling (Vennell, 2012). The friction c_t from the turbine parameterisation is computed from the turbine positions and a turbine radius of $r = 10$ m with equation (5.4). Finally, the density of water $\rho = 1,000\ \mathrm{kgm^{-3}}$ is used in the functionals of interest (5.5) and (5.6).



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

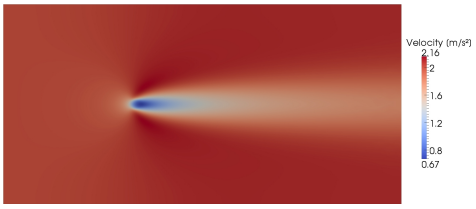
Figure 5.5.: The three turbine layout domains considered in the numerical examples, motivated by Draper et al. (2010). The dashed lines mark the 320 m $\times$ 160 m sized sites where 32 turbines are to be deployed. In the first two domains a constant inflow velocity is enforced, while the third domain is driven by a sinusoidal inflow.

Scenarios 1 and 2 are modelled using the stationary shallow water equations with the following boundary conditions. On the inflow boundary a constant inflow velocity of 2 m/s is enforced and on the outflow boundary the free-surface displacement is set to zero. A no-normal flow boundary condition is applied on the remaining boundaries. The third scenario solves the non-stationary shallow water equations with boundary conditions explained below.

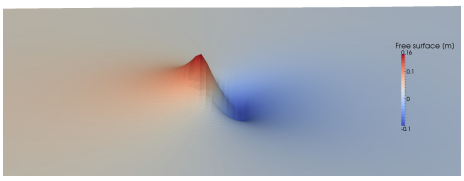
All examples use unstructured meshes with a uniform mesh element size of $h = 20$ m outside the site area. Inside the site area, the mesh is structured with an element size of $h = 2$ m. The higher resolution in the turbine site ensures that each individual turbine is well resolved. Doubling the resolution for the problem considered in section 5.6.1 changed the functional value by less than 0.5%. For the purpose of the numerical examples it is therefore assumed that the problem is sufficiently well resolved. The resulting meshes



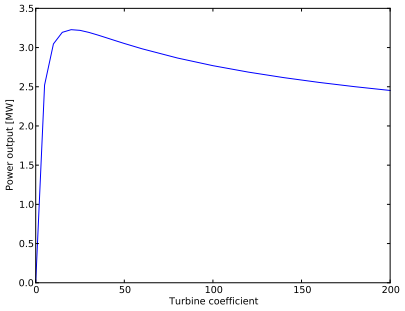
(a) The turbine friction



(b) The velocity magnitude



(c) Free-surface displacement around the turbine



(d) The extracted energy from the flow plotted against the friction coefficient K in equation (5.3)

Figure 5.6.: The setup and results of deploying a single turbine in the domain of scenario 1.

consist of 31,600 triangles for scenario 1, 172,339 triangles for scenario 2 and 161,812 triangles for scenario 3.

The optimisation algorithm is initialised by deploying the 32 turbines in a regular 8×4 grid. Box constraints for the turbine positions are used to ensure that the turbines remain inside the marked areas, see section 5.3.4.

5.6.1. A single turbine

As a preliminary test a single turbine is deployed in the setup of scenario 1. The turbine is placed 213.3 m $\times$ 160 m away from the left bottom corner, as shown in figure 5.6a.

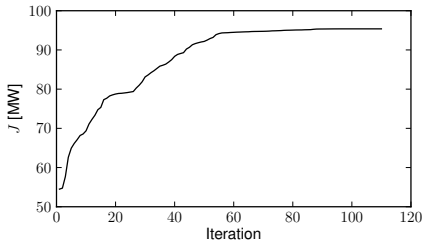
This setup is used to get insight into how the turbine performance depends on the scaling factor K in the turbine friction definition (5.3). The energy extracted from the turbine was computed for a range of K values. The results are plotted in figure 5.6d. The graph shows a defined single peak where the energy extraction is maximised, and is similar to previous studies (Vennell, 2011, 2012). Too high K values deflect the flow around the

turbine and result in the observed power drop. An optimisation algorithm was used to determine that the associated scaling factor is approximately $K = 21$. With that choice the turbine extracts 3.23 MW from the flow. Note that this energy does not represent the actual electrical energy generation, since it does not consider losses due to the turbine support structures and the conversion to electricity. This scaling factor is used for all the following numerical tests if not otherwise stated.

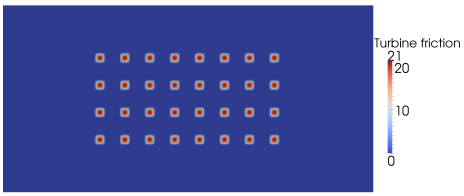
5.6.2. Scenario 1

First, the layout problem for scenario 1 (see figure 5.5a) was solved without the inequality constraints described in section 5.3.4. That is, the turbines can be placed arbitrarily inside the site area and may even overlap. With that setup the optimisation algorithm terminates after 119 iterations and requires 110 gradient and 186 functional evaluations. The results are shown in figure 5.7. Compared to the initial layout shown in figure 5.7b, the optimisation algorithm increases the extracted energy by 75% from 54.5 MW to 95.4 MW (figure 5.7a). In the optimised layout, shown in figure 5.7c, the turbines are aligned in the shape of two $\supset$s with the open end facing the inflow. An interpretation of this layout is that the fluid gets trapped by the sides of the $\supset$s and then pushed through the dense wall of turbines at its closed end. This interpretation is confirmed by the increasing free-surface displacement difference along the sides of the $\supset$s and the large jump along their closed end, see figure 5.7d. Similarly, figure 5.7e shows a stepwise reduction of the velocity magnitude where the turbines are aligned perpendicularly to the flow direction.

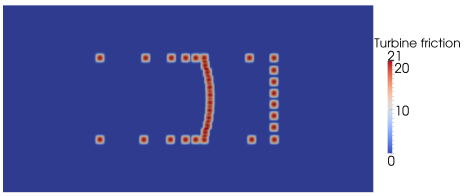
In the next setup, the inequality constraints are included to enforce a minimum distance of 3 turbine radii (i.e. 30 m) between each turbine. With this setup, the optimisation algorithm terminates after 64 iterations with 63 gradient and 90 functional evaluations. The energy extracted increases by 40% from 54.5 MW to 76.1 MW. The reduction of the optimised energy extraction compared to the previous setup is expected because the inequality constraints add further restrictions to the feasible turbine positioning. In particular, the previous optimised turbine positioning is not a feasible solution in this setup. The optimised alignment, shown in figure 5.8c, differs significantly from the previous solution. The two main characteristic struc-



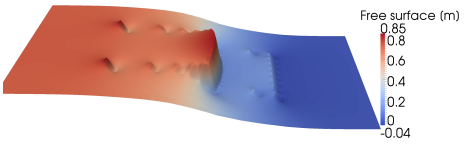
(a) The extracted energy



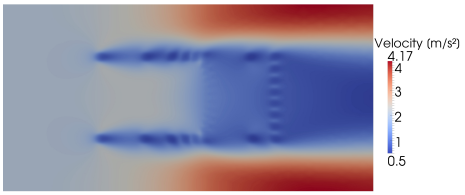
(b) Initial turbine positions



(c) Optimised turbine positions



(d) Free-surface displacement



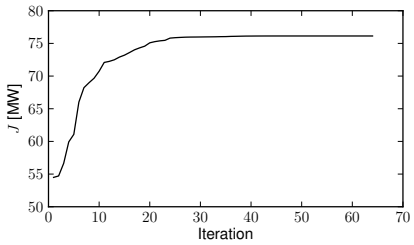
(e) Velocity magnitude

Figure 5.7.: Results of scenario 1 with no inequality constraints, i.e. turbines may overlap.

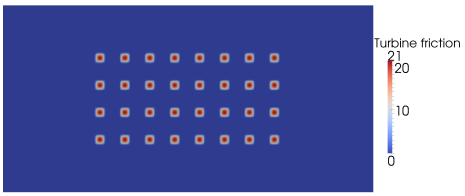
tures are a > shaped alignment close to the inflow boundary and a wall of turbines aligned perpendicular to the flow near the outflow boundary. Also staggering can be seen in the interior of the site where one turbine is never directly behind another turbine. Compared to the previous setup, the free-surface displacement and the velocity magnitude decrease more gradually towards the outflow, see figures 5.8d and 5.8e.

5.6.3. Scenario 2

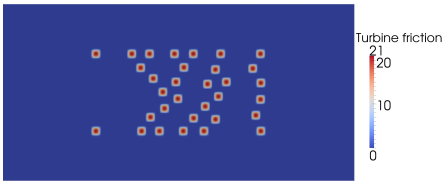
The domain of the second scenario is shown in figure 5.5b. The inequality constraints from the previous scenario are applied to this scenario to enforce a minimum distance between the turbines. The optimisation algorithm



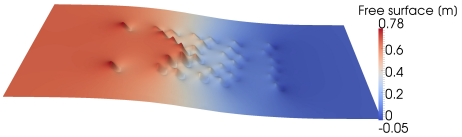
(a) The extracted energy



(b) Initial turbine positions



(c) Optimal turbine positions



(d) Free-surface displacement

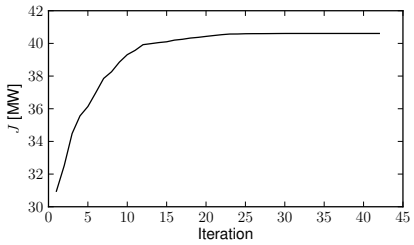


(e) Velocity magnitude

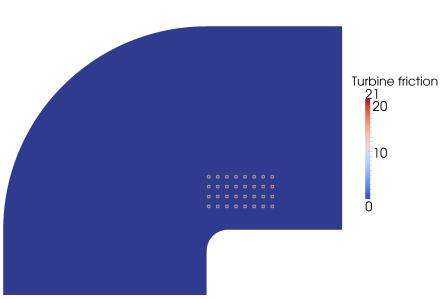
Figure 5.8.: Results of scenario 1 with inequality constraints.

terminates after 42 iterations and requires 41 gradient and 60 functional evaluations. The total power output increases by 31% from 30.9 MW to 40.6 MW, see figure 5.9a. The optimised positioning, shown in figure 5.9b features a distinct diamond shaped turbine layout with a hole on the inflow facing side. Figures 5.9d and 5.9e of the free-surface displacement and velocity magnitude, respectively, suggest that this hole acts to trap and push the flow through the down-flow positioned turbines similar to the previous examples.

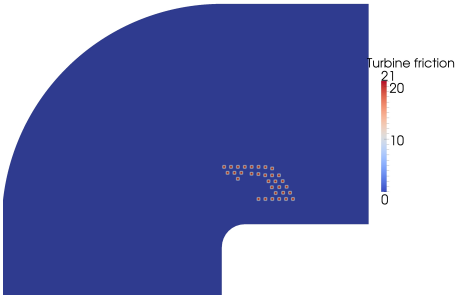
This scenario is also used to investigate the potential of varying the scaling factor K in the turbine parameterisation (5.3). For that, the optimisation problem is altered such that both the turbine positions and the K values for each turbine are included as optimisation variables. Additionally, box



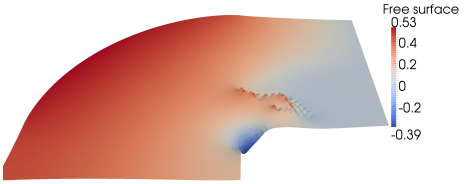
(a) The extracted energy



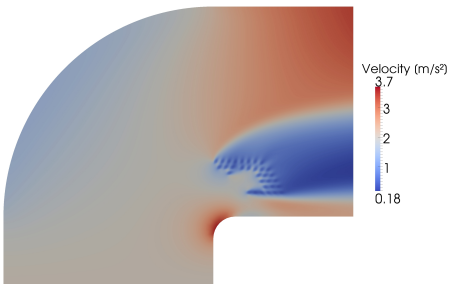
(b) Initial turbine positions



(c) Optimal turbine positions



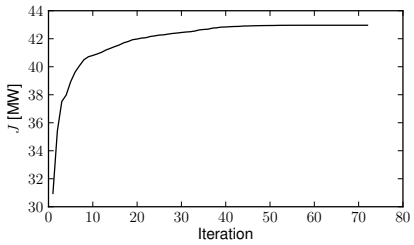
(d) Free-surface displacement



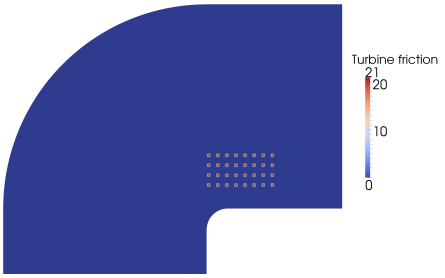
(e) Velocity magnitude

Figure 5.9.: Results of scenario 2 with inequality constraints.

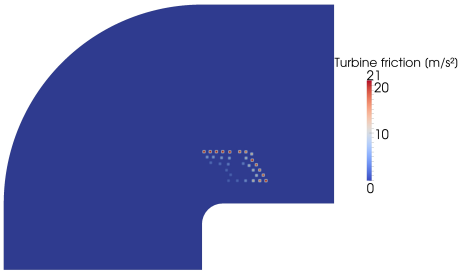
constraints are added to ensure that $0 \leq K \leq 21$ for each turbine. With this setup, the optimisation algorithm terminates after 72 iterations and requires 71 gradient and 118 functional evaluations. The results are presented in figure 5.10. Compared to the initial layout, the energy extraction increases by 39% from 30.9 MW to 43.0 MW - the additional freedom of the optimisation algorithm has resulted in a higher optimum energy extraction. The optimised turbine layout is similar in shape to the previous solution but with a less distinct hole on the inflow facing side. However, the friction coefficients of most turbines are significantly reduced. Only the turbines on the down flow edges of the diamond take the maximum K value. Despite



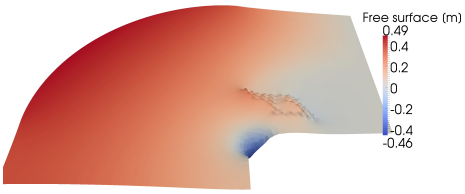
(a) The extracted energy



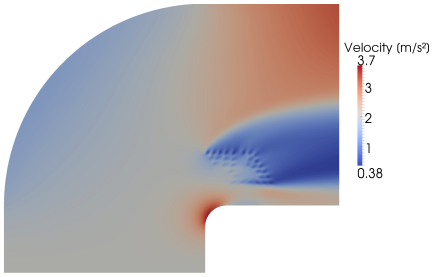
(b) Initial turbine friction



(c) Optimal turbine friction



(d) Free-surface displacement



(e) Velocity magnitude

Figure 5.10.: Results of scenario 2 with inequality constraints and optimising for the turbine scaling factor as well as for the turbine positions.

this reduction, this optimised layout extracts about 8% more energy compared to the solution of the previous setup. This suggests that reducing the size of some of the turbines or removing them entirely might lead to a better cost-performance ratio. In general this example shows that by controlling not only the position of the turbines but also their scaling factor (which can be viewed as controlling the pitch of the turbine blades) can lead to a significant increase in the energy output.

5.6.4. Scenario 3

The last example is modelled by the non-stationary shallow water equations in the domain shown in figure 5.5c. The setup optimises only the turbine positions and inequality constraints are used to enforce a minimum turbine distance of 30 m. No-normal flow boundary conditions are enforced on the sides and on the island. Dirichlet boundary conditions are applied on the left and right boundary of the domain to enforce the following sinusoidal in-/outflow velocity:

$$u(x, y, t) = \begin{pmatrix} -2\sin(2\pi t/P) \\ 0 \end{pmatrix},$$

with a $P = 12$ h period. The simulation time consists of one full sinusoidal period and the time step was set to $\Delta t = 864$ s, corresponding to a total of 50 time steps per simulation.

The results for this optimisation setup are show in figure 5.11. With this setup, the optimisation algorithm terminates after 57 iterations and requires 56 gradient and 86 functional evaluations. The averaged energy extraction of the turbines increases by 22% from 48.4 MW to 59.0 MW. Since the computational domain is symmetric and the simulation time covers one full period, the optimal layout is expected to be symmetric in the x-direction. The numerical solution, shown in figure 5.11b, shows indeed an almost symmetric solution. The turbine alignment consists of two distorted v shapes whose open ends face the in/-outflow boundaries. Similar to the previous example, an interpretation for this alignment is to divert the stream towards the corner of the v where turbines can extract large amounts of energy. An additional row of turbines can be seen parallel to the bottom of the domain. These turbines are positioned to capture energy from the flow passing along the boundary.

5.7. Conclusions

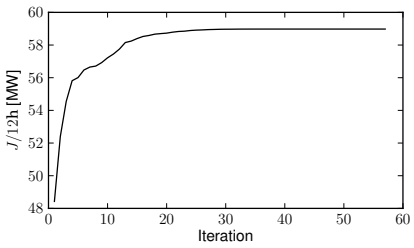
The optimal positioning of tidal turbines in a tidal stream was formulated as an optimisation problem constrained by the shallow water equations. This problem was then solved with the optimisation framework from chapter 4

using sequential quadratic programming and the adjoint approach for the gradient computations.

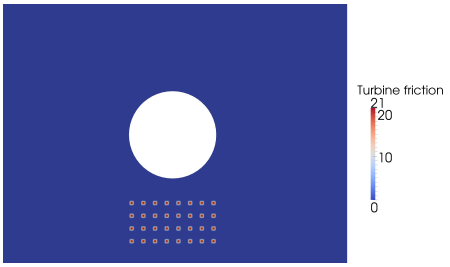
The positioning problem was applied to four different idealised scenarios. For each of these, the optimisation algorithm was able to increase the total power extraction by between 22% to 75% compared to the initial layout. The resulting optimal layouts show the consistent feature of a "V" shaped alignment of the turbines, that to the knowledge of the author has not been proposed before.

More tests have been performed that are not shown in this study. For example, by relaxing the box constraints to allow the turbines to be positioned in the whole domain, the optimised solutions consists of tightly packed turbines forming "barrages" across the channel. Similar alignments have been suggested in the literature, for example by Salter (2009).

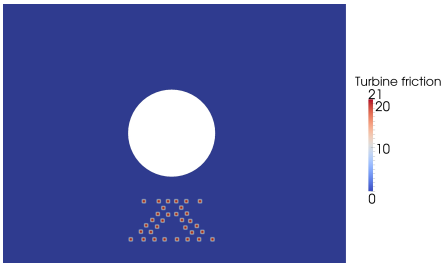
The source-code of the turbine layout optimisation program and all examples are open-source and available at `http://opentidalfarm.org`.



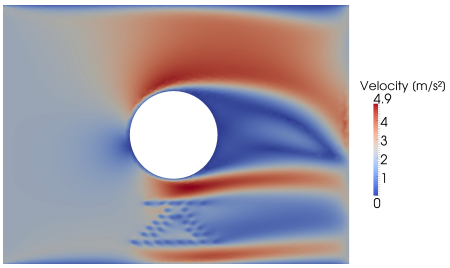
(a) The averaged energy extraction



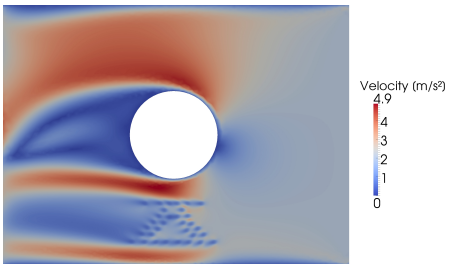
(b) Initial turbine friction



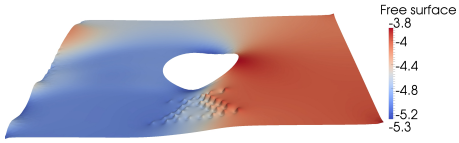
(c) Optimal turbine friction



(d) Velocity magnitude at the time when the input velocity from the left reaches its maximum



(e) Velocity magnitude at the time when the input velocity from the right reaches its maximum



(f) The associated free-surface displacement of figure 5.11e

Figure 5.11.: Results of the non-stationary scenario 3.

Chapter 6

Data assimilation for the shallow water equations with wetting and drying

Contents

6.1. Introduction 145

6.2. Shallow water model with wetting and drying . 146

6.3. The optimisation problem 156

6.4. Numerical examples 158

6.5. Summary 166

Abstract

This chapter applies data assimilation to inundation problems governed by the non-linear shallow water equations with wetting and drying. The objective is to minimise the misfit between the simulated and an observed wet/dry interface by recovering the wave profile at the inflow boundary. The resulting optimisation problem is solved with gradient-based optimisation using the framework described in chapter 4. The capabilities of this approach are demonstrated on an idealised sloping beach setup in which the profile of an incoming wave is reconstructed from wet/dry interface observations. Furthermore, the wave profile reconstruction from inundation observations of a laboratory experiment

of the Hokkaido-Nansei-Oki tsunami is considered. In both examples, the wave profile is determined with an error of less than 1% of the correct wave signal.

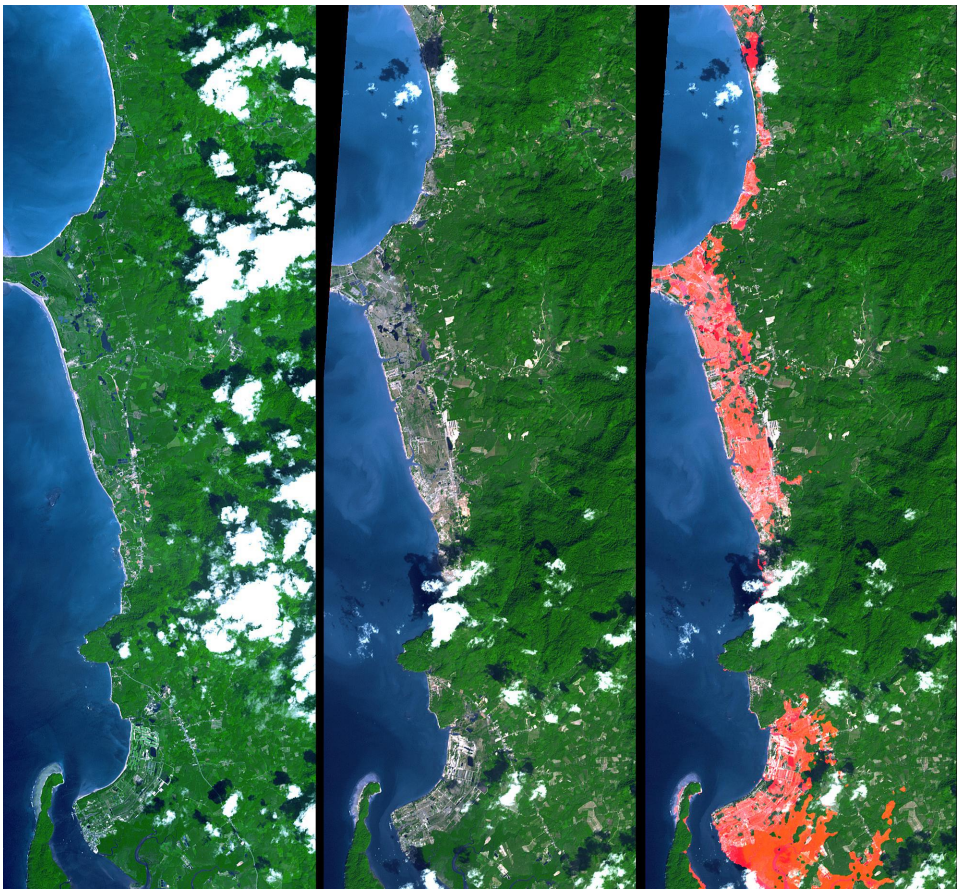


Figure 6.1.: Tsunami inundation, North of Phuket, Thailand. Is it possible to reconstruct the incoming tsunami wave (and potentially the mechanics of the source) from such images?
Image credit: ASTER: NASA/GSFC/METI/ERSDAC/JAROS, and U.S./Japan ASTER Science Team SRTM: NASA/JPL/NGA.

6.1. Introduction

Wetting and drying plays an important role in coastal research to study tsunami waves (Kowalik et al., 2005), tidal flats and river estuaries (Zhang et al., 2009; Xue and Du, 2010; Kärnä et al., 2011), and flooding events (Westerink et al., 2008; Song et al., 2011). For the simulation of the wetting and drying processes many algorithms have been proposed, both for the shallow-water equations (Medeiros and Hagen (2012) and the references therein) and for the Navier-Stokes equations (Funke et al., 2011).

In addition to the simulation of these problems, it is often desired to study the sensitivity of the solution with respect to input parameters such as initial and boundary conditions. The key for the efficient computation of these sensitivities is the adjoint approach (see chapter 1). In the context of shallow water modelling without wetting and drying, adjoint models have been successfully used in various applications, ranging from data assimilation (Bagchi and Brummelhuis, 1994; Gejadze and Copeland, 2005; Chen and Navon, 2009) and parameter identification (Ding and Wang, 2005) to wave and flood control (Kawahara and Kawasaki, 1990; Sanders and Katopodes, 1996, 2000; Ding and Wang, 2006; Samizo and Kawahara, 2011). Blaise et al. (2012) successfully reconstructed the initial condition of tsunami simulations from buoy measurements, but also emphasized the importance of including wetting and drying in the adjoint model as future work.

The main contribution of this chapter is the development of an adjoint model for the shallow water equations with wetting and drying to compute the sensitivity of the wet/dry interface with respect to boundary conditions. The sensitivity information is then used in combination with gradient-based optimisation to solve data assimilation problems in an efficient manner. The considered problems aim to find the boundary condition values that lead to an observed wet/dry interface movement over time.

6.2. Shallow water model with wetting and drying

6.2.1. Continuous formulation

The non-linear shallow water equations with appropriate initial and boundary conditions are:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u + g\nabla\eta = \frac{c_t(H)}{H}\|u\|u \qquad \text{in } \Omega \times (0,T), \tag{6.1a}$$

$$\frac{\partial \eta}{\partial t} + \nabla \cdot (Hu) = 0 \qquad \text{in } \Omega \times (0,T), \tag{6.1b}$$

$$u \cdot n = 0 \qquad \text{on } \partial\Omega_S \times (0,T), \tag{6.1c}$$

$$\eta = \eta_D \qquad \text{on } \partial\Omega_D \times (0,T), \tag{6.1d}$$

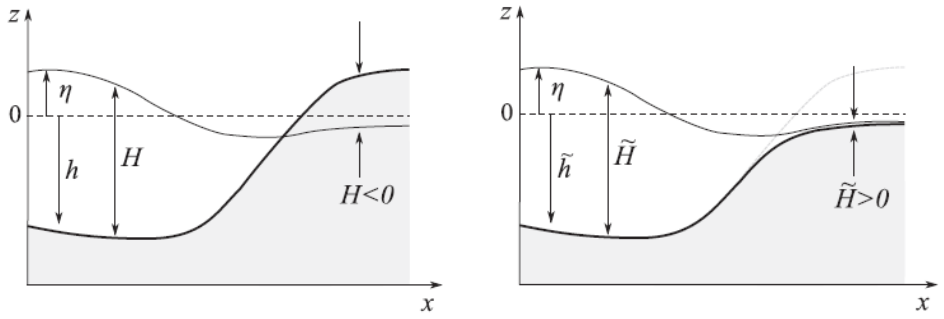
$$u = u_0, \quad \eta = \eta_0 \qquad \text{at } \Omega \times \{0\}, \tag{6.1e}$$

where Ω is the domain of interest, T is the final time, u is the unknown depth-averaged velocity, η is the unknown free-surface displacement, h describes the static bathymetry, $H = \eta + h$ is the total water depth, u_0 and η_0 are the initial conditions, and n is the normal vector on the boundary. The water height variables are visualised in figure 6.2a. The domain boundary is divided into $\partial\Omega_S$, where a no-normal flow condition is imposed, and $\partial\Omega_D$, where a Dirichlet boundary condition prescribes the free-surface displacement η_D. The remaining parameters are the gravitational force g and the friction coefficient in the Chézy-Manning formulation:

$$c_t(H) = \frac{g\mu^2}{H^{1/3}},$$

where μ is the user specified Manning coefficient.

In its standard form, the shallow water equations do not account for wetting and drying processes. Various extensions have been proposed and are summarised in Medeiros and Hagen (2012). A classical approach is to mark individual mesh elements in the computational domain as wet or dry and remove dry elements from the time step computation. However, the elemental wet/dry conditions are usually discontinuous functions, which complicates the development of the adjoint system. This can be seen in the work of Miyaoka and Kawahara (2008), where the wetting and drying algorithm was entirely ignored in the adjoint computation; instead, the adjoint shallow water equations without wetting and drying were solved



(a) Variables for the static bathymetry (b) Variables for the dynamic bathymetry

Figure 6.2.: Modified water depth variables, source Kärnä et al. (2011).

only in the wet area. Such an approach does not provide the sensitivity of the wet/dry interface, which is crucial for the applications considered here. This work is based on an alternative wetting and drying algorithm proposed by Kärnä et al. (2011), motivated by the fact that the authors use the Newton method to solve the discretised equations, hence ensuring the differentiability of the numerical schemes.

The wetting and drying algorithm developed by Kärnä et al. (2011) is based on the idea to replace the static bathymetry h with a dynamic bathymetry $\tilde{h}$, which moves such that the water level remains always positive. This dynamic bathymetry is defined as:

$$\tilde{h}(x,t) \coloneqq h + f(H),$$

where f is a smooth function that ensures the positiveness of the total water depth, that is:

$$\tilde{H} \coloneqq \eta + \tilde{h} > 0. \tag{6.2}$$

The resulting variables for the dynamic bathymetry approach are visualised in figure 6.2b. For the function f Kärnä et al. (2011) suggests a smooth approximation of the maximum operator:

$$f(H) \coloneqq \frac{1}{2}\left(\sqrt{H^2+\alpha^2} - H\right) \approx \max(0,-H). \tag{6.3}$$

This function choice, plotted in figure 6.3a, is also used in this work. The constant parameter α in definition (6.3) controls the accuracy of the approx-

imation to the non-differentiable max operator. Kärnä et al. (2011) gives a guideline for determining a suitable estimate for this constant:

$$\alpha \approx \epsilon \coloneqq d_e \|\nabla h\|, \tag{6.4}$$

where d_e is a typical length scale of the mesh element.

The modified shallow water equations that include wetting and drying are obtained from the original equations (6.1) by replacing the total depth H with its dynamic variant $\tilde{H}$ and including the time derivative of the dynamic bathymetry $\tilde{h}$ in the continuity equation to account for the temporal variability of the bathymetry:

$$\begin{aligned}
\frac{\partial u}{\partial t} + (u \cdot \nabla)u + g\nabla\eta &= \frac{c_t(\tilde{H})}{\tilde{H}}\|u\|u && \text{in } \Omega \times (0,T), \\
\frac{\partial \eta}{\partial t} + \frac{\partial \tilde{h}}{\partial t} + \nabla \cdot (\tilde{H}u) &= 0 && \text{in } \Omega \times (0,T), \\
u \cdot n &= 0 && \text{on } \partial\Omega_S \times (0,T), \\
\eta &= \eta_D && \text{on } \partial\Omega_D \times (0,T), \\
u = u_0, \quad \eta &= \eta_0 && \text{at } t = 0.
\end{aligned} \tag{6.5}$$

Finally, to avoid non-differentiable functions in the continuous formulation, the norm operator in (6.5) is replaced by a smooth approximation:

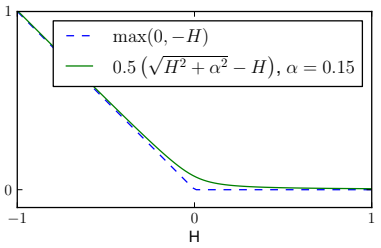
$$\|u\| \approx \sqrt{\|u\|^2 + \alpha^2},$$

with the same α constant as above. A plot of this approximation function is given in figure 6.3b.

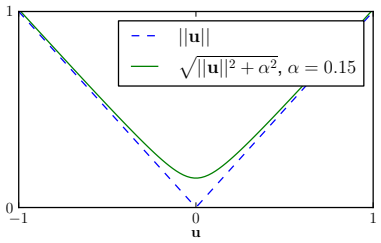
6.2.2. Spatial discretisation

The modified shallow water equations (6.5) are discretised in space with the mixed continuous-discontinuous finite element method. A general introduction to discontinuous Galerkin methods can be found in Hesthaven and Warburton (2008).

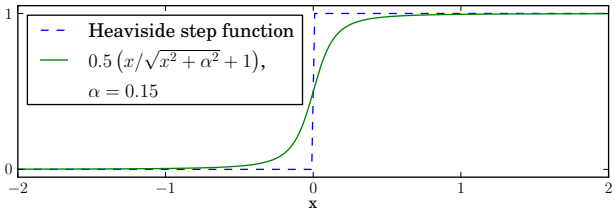
The discrete function spaces are constructed with the $\mathrm{P1_{DG}}$-P2 finite element pair (Cotter et al., 2009; Comblen et al., 2010). Let V and W denote the associated function spaces for the velocity and free-surface displacement, respectively. The weak formulation is obtained by multiplying



(a) Smooth approximation of the function $\max(0, -H)$, which is used to enforce a positive water level in equation (6.2)



(b) Smooth approximation of the norm function that occurs in the drag term of the momentum equation (6.1a)



(c) Smooth approximation of the Heaviside step function (6.10) used in the functional of interest as an indicator function for dry areas. Note that the smooth representation is equivalent to $f'(x) + 1$, where f is defined in equation (6.3)

Figure 6.3.: Smooth approximations of the non-differentiable functions that occur in the problem formulation. This chapter uses the same smoothness constant α for all approximations.

equations (6.5) with test functions $\Psi \in V$ and $\Phi \in W$ and integrating over the domain Ω. A weak incorporation of the boundary conditions leads to following formulation: find $u \in V, \eta \in W$ such that $\forall\ \Psi \in V,\ \Phi \in W$:

$$\left\langle \frac{\partial u}{\partial t}, \Psi \right\rangle_\Omega + \langle (u \cdot \nabla) u, \Psi \rangle_\Omega - \sum_{e \in E} \left(\left\langle \{u^+\} [\![u]\!], \Psi^+ \right\rangle_e - \left\langle \{u^-\} [\![u]\!], \Psi^- \right\rangle_e \right)$$
$$+ \langle g \nabla \eta, \Psi \rangle_\Omega - g \langle \eta - \eta_D, \Psi \cdot n \rangle_{\partial\Omega_D} = \left\langle \frac{c_t(\tilde{H})}{\tilde{H}} \|u\| u, \Psi \right\rangle_\Omega, \tag{6.6a}$$

$$\left\langle \frac{\partial \tilde{H}}{\partial t}, \Phi \right\rangle_\Omega - \left\langle (\tilde{H} u), \nabla \Phi \right\rangle_\Omega + \left\langle \tilde{H} u \cdot n, \Phi \right\rangle_{\partial\Omega \setminus \partial\Omega_S} = 0. \tag{6.6b}$$

Here, E denotes the interior mesh facets and the superscripts $+$ and $-$ are used to distinguish between the two facet values of the discontinuous function. $\{u\}$ represents the downwind value of u, i.e.:

$$\{u\} := \begin{cases} u \cdot n & \text{if } u \cdot n < 0, \\ 0 & \text{otherwise,} \end{cases}$$

and $[\![u]\!]$ denotes the jump of u across the facet side:

$$[\![u]\!] := u^+ - u^-.$$

A simple upwinding scheme is implemented for the advection term, which is obtained by integrating the advection term by parts, replacing the advected velocity at the inflow facets with the upwind velocity and then integrating by parts again. The no-normal flow boundary condition is weakly applied by neglecting the surface integrals associated with the domain boundary $\partial\Omega_S$ in equation (6.6b). Similarly, the pressure term in the momentum equation (6.6a) is integrated twice by parts to weakly enforce the Dirichlet boundary condition on $\partial\Omega_D$.

As discussed in Kärnä et al. (2011), volume conservation is only satisfied if the integrals featuring the continuity equation (6.6b) are evaluated accurately. However, $\tilde{H}$ is not a polynomial function and therefore standard quadrature rules would require an infinite degree to exactly evaluated these integrals. Section 6.2.4 investigates this issue and shows how a finite quadrature degree affects the volume conservation. Another difficulty is to ensure that $\tilde{H}$ is everywhere positive on the discrete level. Kärnä et al. (2011) uses

piecewise linear elements for $\tilde{H}$ and exploits the fact that functions based on linear finite elements take their extrema at vertices. Therefore a nodewise projection for $\tilde{H}$ ensures a domain-wide positive water level. To circumvent this problem for the quadratic elements used here, $\tilde{H}$ itself is never stored as a discrete function, but is instead reevaluated for each quadrature point using equation (6.2).

6.2.3. Temporal discretisation

Following Kärnä et al. (2011), the weak equations are discretised in time using the second-order Diagonally Implicit Runge-Kutta (DIRK22) scheme (Ascher et al., 1997, §2.6). It is an implicit time stepping scheme and therefore allows for large time steps in the time integration.

For brevity, the weak equations (6.6) are written in the shortened form:

$$\left\langle \frac{\partial u}{\partial t}, \Psi \right\rangle_\Omega = S_u(\eta, u),$$
$$\left\langle \frac{\partial \tilde{H}}{\partial t}, \Phi \right\rangle_\Omega = S_\eta(\eta, u).$$

To begin with, the continuous time period is split into discrete levels with associated time steps Δt. For each time level, DIRK schemes solve a sequence of stages, each of which consists of solving a non-linear equation. Let the superscript n denote the time level and a superscript i denote the stage. The computation of time level n involves following steps:

- For each stage $i = 1, \ldots, s$ solve following non-linear system to obtain the intermediate solutions u^i and η^i:

$$\left\langle u^i, \Psi \right\rangle_\Omega = \left\langle u^{n-1}, \Psi \right\rangle_\Omega + \Delta t \sum_{j=1}^{i} a_{i,j} S_u(\eta^j, u^j),$$
$$\left\langle \tilde{H}^i, \Phi \right\rangle_\Omega = \left\langle \tilde{H}^{n-1}, \Phi \right\rangle_\Omega + \Delta t \sum_{j=1}^{i} a_{i,j} S_\eta(\eta^j, u^j).$$

  Each stage has an associated time level of $t^i = t^n + c_i \Delta t$ which is used to compute the values of forcing terms. The coefficients $a_{i,j}$ and c_i depend on the specific Runge-Kutta method and are defined below.

- The final stage linearly combines these intermediates solutions to obtain the unknowns at the next time level u^n and η^n:

$$\langle u^n, \Psi \rangle_\Omega = \langle u^{n-1}, \Psi \rangle_\Omega + \Delta t \sum_{j=1}^{s} b_j S_u(\eta^j, u^j),$$
$$\left\langle \tilde{H}^n, \Phi \right\rangle_\Omega = \left\langle \tilde{H}^{n-1}, \Phi \right\rangle_\Omega + \Delta t \sum_{j=1}^{s} b_j S_\eta(\eta^j, u^j).$$

  Again, the coefficients b_j depend on the specific Runge-Kutta method used.

In general, the Runge-Kutta coefficients a_{ij}, b_j and c_i are defined compactly in the form of a Butcher tableaux:

c_1	$a_{1,1}$			
c_2	$a_{2,1}$	$a_{2,2}$		
$\vdots$	$\vdots$	$\vdots$	$\ddots$	
c_s	$a_{s,1}$	$a_{s,2}$	$\dots$	$a_{s,s}$
	b_1	b_2	$\dots$	b_s

The Butcher tableaux of the DIRK22 scheme used in this work is defined as (Ascher et al., 1997, §2.6):

γ	γ	
1	$1-\gamma$	γ
	$1-\gamma$	γ

with $\gamma \coloneqq \left(2 - \sqrt{2}\right)/2$.

6.2.4. Verification

The discretised shallow water equations with wetting and drying was implemented and solved with the PDE-framework FEniCS (Logg et al., 2011). To verify the implementation, an analytical test case published by Thacker (1981) is used. This test case considers an undamped wave in a flat, bowl shaped basin where wetting and drying occurs on its sides.

The test domain consists of a circular basin with parabolic depth defined as:

$$h(x, y) \coloneqq h_c \left(1 - \frac{x^2 + y^2}{L^2}\right),$$

where L and h_c are positive constants describing the basin's length and depth at its centre, respectively. The analytical solution solves the shallow water equations without bottom friction, that is $\mu = 0$, and is (Thacker, 1981):

$$u_{\text{exact}}(x, y, t) \coloneqq \frac{\omega A \sin(\omega t)}{2(1 - A\cos(\omega t))} \begin{pmatrix} x \\ y \end{pmatrix},$$
$$\eta_{\text{exact}}(x, y, t) \coloneqq h_c \left(\frac{\sqrt{1 - A^2}}{1 - A\cos\omega t} - 1 - \frac{x^2 + y^2}{L^2} \left(\frac{1 - A^2}{(1 - A\cos\omega t)^2} - 1 \right) \right),$$

where

$$\omega^2 := \frac{8gh_c}{L^2}, \qquad A := \frac{(h_c + \eta_c)^2 - h_c^2}{(h_c + \eta_c)^2 + h_c^2},$$

and η_c is the maximum free-surface displacement at the basin's centre. The parameters for the numerical tests were chosen consistently with Balzano (1998): $L = 430.62$ km, $h_c = 50$ m, $\eta_c = 2$ m and a gravity magnitude of $g = 9.81$ m/s^2. This results in a periodic free-surface oscillation with a 12 h period, see figure 6.4.

The Thacker test case was numerically solved on four meshes with increasing resolution (figure 6.5). To ensure that the domain is sufficiently large to capture the wetting and drying process, the computational domain consists of a circle with radius 496.20 km, in accordance to Kärnä et al. (2011). The simulation was carried out for 24 h with a time step of 300 s. This time step is small enough to ensure that the spatial error dominates the temporal discretisation error: performing the convergence analysis with a time step of 150 s resulted in similar convergence results. The smoothness constant α is estimated using equation (6.4) and yields $\alpha \approx 2.4$ m for the finest mesh. Numerical experiments showed that this value can further be reduced to $\alpha = 1.8$ m without loosing the stability of the simulation. Hence, this reduced value was used for the finest mesh, and linearly increased with the mesh element sizes for the coarser meshes (i.e. $\alpha = 1.8, 3.6, 5.4, 7.2$ m for the $10, 20, 30, 40$ km meshes).

The model implementation was verified by repeating the convergence test performed by Kärnä et al. (2011) and comparing the resulting order of

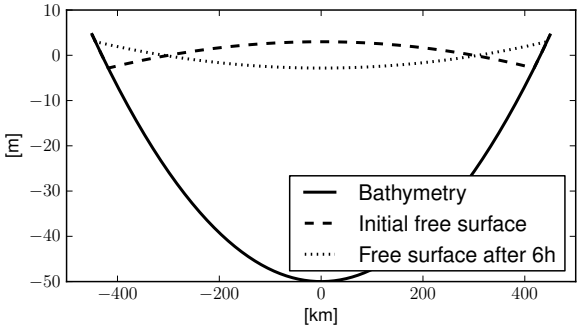
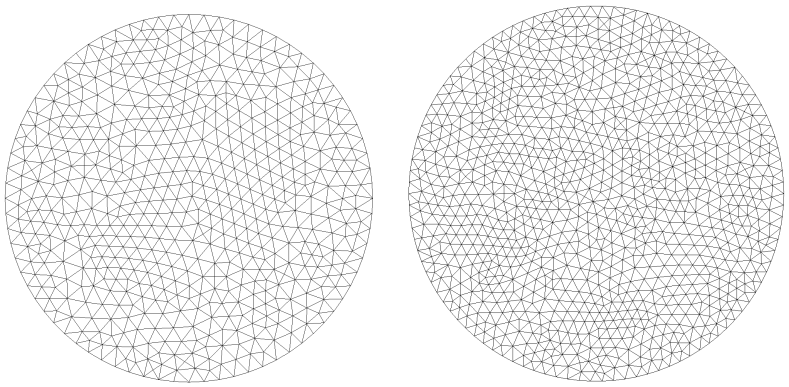
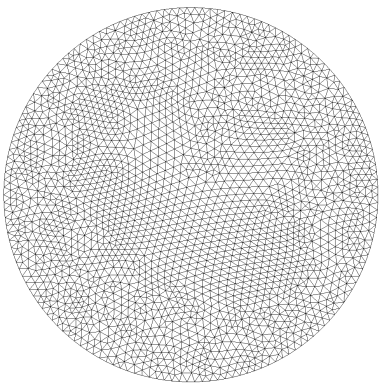


Figure 6.4.: The setup of the Thacker problem. The free-surface oscillates with a 12 h period, while wetting and drying occurs at the sides of the basin.

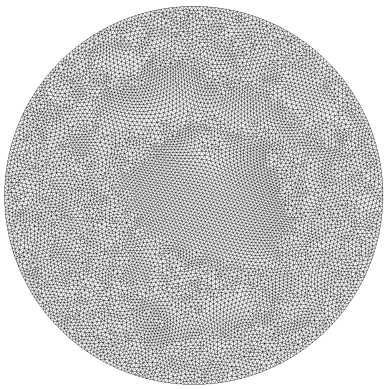


(a) 40 km mesh resolution

(b) 30 km mesh resolution



(c) 20 km mesh resolution



(d) 10 km mesh resolution

Figure 6.5.: The four meshes (generated with Gmsh (Geuzaine and Remacle, 2009)) used for the Thacker test case to determine the spatial convergence rate of the shallow water model with wetting and drying.

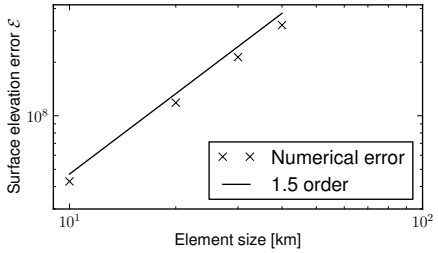


Figure 6.6.: Spatial discretisation errors computed from the four meshes shown in figure 6.5. The average rate of convergence is 1.46, Kärnä et al. (2011) observed an order of 1.47.

convergence. The error measure is defined as:

$$\mathcal{E} := \sqrt{\langle \tilde{\eta} - \tilde{\eta}_{\text{exact}}, \tilde{\eta} - \tilde{\eta}_{\text{exact}} \rangle_{\Omega \times (0,T)}},$$

where $\tilde{\eta} := \tilde{H} - h$ and $\tilde{\eta}_{\text{exact}} := \max(\eta_{\text{exact}}, -h)$ are the numerical and analytical solutions that take the bathymetry into account. The numerical errors for the four meshes are plotted in figure 6.6. The average convergence rate is 1.46 which is close to the convergence rate of 1.47 observed by Kärnä et al. (2011).

6.2.5. Volume conservation

As discussed in section 6.2.2, it has to be assessed whether the inaccurate quadrature computations affect the volume conservation. Furthermore, the non-linear systems are solved using a Newton solver whose tolerance settings might impact the volume conservation as well.

To test the volume conservation, the Thacker test case introduced in the previous section was used. It consists of a closed domain and therefore the fluid volume must remain constant throughout time. Using the setup with the coarsest mesh from the previous section, the Thacker test case was solved for a combination of different quadrature degrees and Newton tolerances. For each combination, the maximum relative error in the volume conservation was computed as:

$$\mathcal{E}_V := \max_{t \in (0,T)} \left| \frac{V(0) - V(t)}{V(0)} \right|,$$

Quadrature degree	Newton tolerance	Relative volume error $\mathcal{E}_V$
4	10^{-6}	$1.27 \cdot 10^{-8}$
4	10^{-9}	$5.71 \cdot 10^{-12}$
4	10^{-12}	$4.46 \cdot 10^{-15}$
20	10^{-6}	$1.26 \cdot 10^{-8}$
20	10^{-9}	$5.70 \cdot 10^{-12}$
20	10^{-12}	$2.81 \cdot 10^{-15}$
121	10^{-6}	$1.26 \cdot 10^{-8}$
121	10^{-9}	$5.70 \cdot 10^{-12}$
121	10^{-12}	$2.69 \cdot 10^{-15}$

Table 6.1.: The maximum relative volume conservation error over 24 h on the coarsest mesh (figure 6.5a) for different quadrature degrees and relative Newton solver tolerances.

where $V(t) \coloneqq \int_\Omega \tilde{H}(t)\,\mathrm{d}x$ is the total fluid volume at a time t.

The results of these tests are printed in table 6.1. The volume conservation error is largely dominated by the tolerance of the Newton solver while the quadrature degree has only marginal influence. The following numerical results use a quadrature degree of 20 and a relative Newton solver tolerance of 10^{-9}.

6.2.6. Validation

The validation of the forward model is out of scope of this work. However, the wetting and drying scheme has been applied to the Scheldt estuary and the North sea and validated against tidal stations with good results in Kärnä et al. (2011) and Gourgue (2011).

6.3. The optimisation problem

6.3.1. Mathematical formulation as a PDE-constrained optimisation problem

This section formulates the optimisation problem constrained by the shallow water equations with wetting and drying. In the following section, this formulation is then applied to data assimilation problems.

The functional of interest that is to be minimised measures the misfit between an observed and the simulated wet/dry interface over time. For its

formulation, an indicator function is constructed that approaches 1 in dry and 0 in wet areas. By noting that $\eta \geq h$ in wet and $\eta < h$ in dry areas (see figure 6.2a), this indicator function is defined as $\mathcal{H}(\eta - h)$ where $\mathcal{H}$ denotes following smooth approximation of the Heaviside step function:

$$\mathcal{H}(x) \coloneqq \frac{1}{2}\left(\frac{x}{\sqrt{x^2+\alpha^2}} + 1\right) \approx \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{else,} \end{cases} \tag{6.10}$$

in which the α constant controls the smoothness of the approximation. A plot of this approximation in given in figure 6.3c. With that, the functional of interest is defined as:

$$J(\eta) \coloneqq \frac{1}{2}\left\langle \mathcal{H}(\eta - h) - d, \mathcal{H}(\eta - h) - d \right\rangle_{\Omega\times(0,T)}, \tag{6.11}$$

where d denotes the observed wet/dry interface.

The optimisation parameters consist of the Dirichlet boundary values η_D in the shallow water equations (6.5). For simplicity, it is assumed that these values only vary in time, i.e. are constant in space. Note that the computation of the Runge-Kutta stages requires the Dirichlet boundary values at intermediate time levels, see section 6.2.3. These values are obtained by linearly interpolating the values of the two neighbouring time levels.

At this point, the full optimisation problem can the be stated:

$$\begin{aligned} &\min_{\eta_D,u,\eta} J(\eta) \quad \text{subject to} \\ &\frac{\partial u}{\partial t} + (u\cdot\nabla)u + g\nabla\eta = \frac{c_t(\tilde{H})}{\tilde{H}}\|u\|u && \text{on } \partial\Omega\times(0,T), \\ &\frac{\partial \eta}{\partial t} + \frac{\partial \tilde{h}}{\partial t} + \nabla\cdot(\tilde{H}u) = 0 && \text{on } \partial\Omega\times(0,T), \\ &u\cdot n = 0 && \text{on } \partial\Omega_S\times(0,T), \\ &\eta = \eta_D && \text{on } \partial\Omega_D\times(0,T), \\ &u = u_0, \quad \eta = \eta_0 && \text{at } \Omega\times\{0\}. \end{aligned} \tag{6.12}$$

6.3.2. Adjoint model implementation

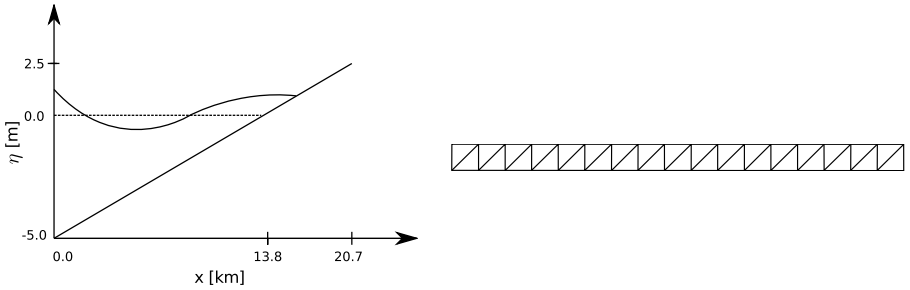
In order to efficiently obtain the solution of the optimisation problem (6.12), the derivative of the functional of interest J with respect to the optimisation parameters η_D is required. This information is computed here using the

adjoint approach described in chapter 1. The continuous adjoint equations of the shallow water equations were discussed in section 5.4.4 and derived in appendix C. In this chapter, the discrete adjoint model is automatically derived using libadjoint and dolfin-adjoint as described in chapters 2 and 3. The necessary code changes were minimal and involved changing less than 10 lines of code. Since all operators in the discrete formulation are differentiable, the resulting functional gradient is consistent with the discrete system. This was tested with the Taylor remainder convergence test, described in section 2.5.6. Its application to a simple example yielded the expected first-order and second-order convergence with and without gradient information, respectively. Furthermore, the Taylor remainder convergence test was successfully applied to the first ten optimisation iterations of the following numerical examples. This gives high confidence that the adjoint system and the gradient computation are correctly implemented. Once the gradient computation was verified the optimisation framework described in chapter 4 was used to solve the optimisation problem (6.12). Again, the development effort was minimal; only two additional lines of code were required.

6.4. Numerical examples

This section presents two applications of the optimisation problem (6.12) where wetting and drying plays an essential role. The problems are solved with the limited memory BFGS (L-BFGS-B) implementation in SciPy (Jones et al., 2001). The L-BFGS-B method belongs to the class of quasi-Newton algorithms that use an approximation of the Hessian matrix based on a limited number of functional gradients, see section 4.3.1. The examples shown in this section use 10 gradients for this approximation.

In order to be able to investigate the effectiveness of the method, the following results are based on synthetically generated observations of the wet/dry interface. These are obtained by first choosing a suitable Dirichlet boundary condition η_D^{exact}, and then executing the forward model while recording the wet/dry interface. Using these records as the observations d in the functional of interest (6.11) guarantees that the chosen Dirichlet boundary condition η_D^{exact} is a solution to the optimisation problem (6.12).



(a) Domain (b) Mesh

Figure 6.7.: The setup for the wave profile reconstruction on a sloping beach.

6.4.1. Wave profile reconstruction on a sloping beach

The first application consists of a long, thin sloping beach with an incoming wave on the deep side. The goal is to reconstruct the wave profile based on observations of the wet/dry interface.

The computational domain is an adaption of a wetting and drying test case considered by Balzano (1998). It consists of a linearly increasing slope of 1.2 km width and 20.7 km length. The left end of the slope is 5 m below, and the right end 2.5 m above the reference level, see figure 6.7a. The Dirichlet boundary condition η_D controls the water level on the left hand side of the domain while a no-normal flow condition is enforced on the other parts of the boundary.

The remaining parameters are $g = 9.81$ m/s^2 for the gravity constant and $\mu = 0.05$ s/m$^{1/3}$ for the Manning drag coefficient. The domain is uniformly discretised with triangular elements of 1.2 km size, see figure 6.7b. The time step is set to $\Delta t = 600$ s and the final time is $T = 24$ h.

The parameters to be optimised are the free-surface displacement values at left boundary for each time level, except for the values during the initial and final 2 h. This avoids the difficulty that, due to the finite wave speed, the Dirichlet boundary values associated with the final time levels have no influence on the wet/dry interface. Similarly, the first Dirichlet boundary value has no influence as it is overwritten by the initial condition of the shallow water equations.

6.4.1.1. Sinusoidal Dirichlet boundary

In this first example, the synthetic Dirichlet boundary η_D^{exact} consists of one sinusoidal wave with $p \coloneqq 12$ h period and 1 m amplitude (figure 6.8b):

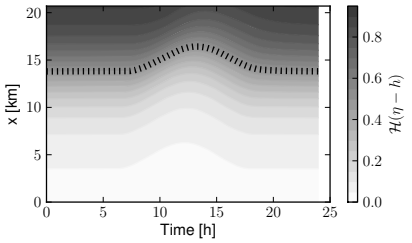
$$\eta_D^{\text{exact}}(t) \coloneqq \begin{cases} 0 & \text{if } t < 6 \text{ h}, \\ \frac{1}{2}\left(-\cos\left(\frac{2\pi(t-6\text{ h})}{p}\right)+1\right) & \text{if } 6 \text{ h} \leq t \leq 18 \text{ h}, \\ 0 & t > 18 \text{ h}. \end{cases} \tag{6.13}$$

The observations d in the functional of interest (6.11) are generated by running the forward model with the synthetic Dirichlet boundary values for 24 h and recording the approximated wet/dry interface $\mathcal{H}(\eta - h)$ at every time level. The resulting observations with a smoothness value of $\alpha = 1.8$ are plotted in figure 6.8a.

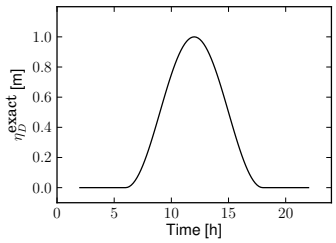
To verify that the synthetic Dirichlet boundary condition η_D^{exact} combined with these observations is indeed a solution to the optimisation problem (6.12), the optimisation algorithm was executed with $\eta_D = \eta_D^{\text{exact}}$ as an initial guess. As expected, the algorithm terminated after the first iteration, reporting that the first-order optimality conditions hold (i.e. the functional gradient is exactly zero).

Next it was tested if the synthetic Dirichlet boundary condition can be recovered without any prior information. For that, the optimisation problem (6.12) was solved with an initial guess of $\eta_D = 0$. The optimisation algorithm was configured to terminate once the norm of the functional gradient decreased below 10^2. With that setup, the algorithm finished after 16 iterations (18 functional evaluations). Figures 6.8b and 6.8c show that the synthetic Dirichlet condition was successfully reconstructed, up to a maximum error of $2.89 \cdot 10^{-6}$ m (figure 6.8d). The remaining error, shown in figure 6.8d, oscillates and is concentrated at the beginning and end of the time period.

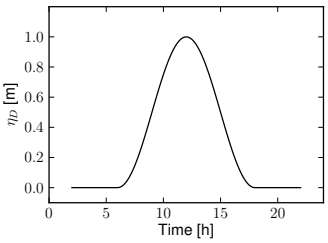
The large smoothing constant used in the previous example can cause unphysical results as described in Kärnä et al. (2011). The guideline equation for the smoothness parameter (6.4) suggests a value of $\alpha = 0.43$. Therefore, the optimisation problem was again solved with this recommended smoothness value. The resulting observations are plotted in figure 6.9a. Compared to the previous example, the sharper gradient at the wet/dry interface can



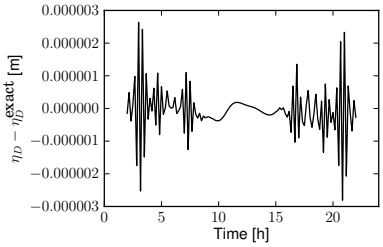
(a) The wet/dry interface observations constructed by running the forward problem with the synthetic Dirichlet boundary values. The observations are approximated indicator functions of the wet/dry interface (marked as dashed line).



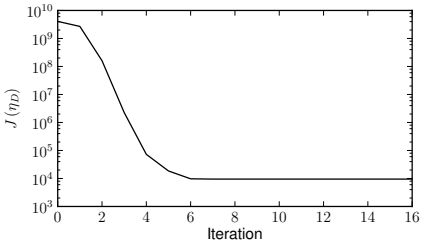
(b) Synthetic Dirichlet boundary values.



(c) Reconstructed Dirichlet boundary values.



(d) Difference between synthetic and reconstructed Dirichlet boundary values.



(e) Functional values during the optimisation iterations.

Figure 6.8.: Results of the wave profile reconstruction on a sloping beach with a sinusoidal Dirichlet boundary and smoothing value $\alpha = 1.8$. The initial and final 2 h of the boundary values are excluded from the reconstruction.

clearly be seen. With this setup, the optimisation algorithm terminates after 93 iterations (99 functional evaluations) and was able to reconstruct the incoming wave profile up to a maximum error of $1.04 \cdot 10^{-5}$ m.

From figure 6.9 it can be seen that the optimisation convergence nearly stagnates after around 10 optimisation iterations. Possible reasons for that are that the optimisation algorithm reaches a point where tolerance settings for the forward or adjoint models become significant, or that regularisation techniques are required to achieve a fast convergence to the optimal solution. A introduction to regularisation techniques can be found in Wang et al. (2010). Nevertheless, the reconstruction error is of order 0.001% of the wave height, which is sufficiently accurate for many practical applications.

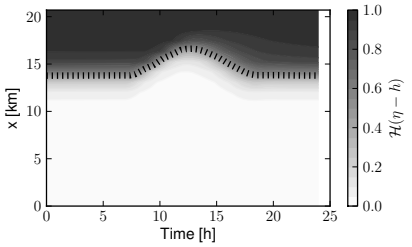
6.4.1.2. Composed sinusoidal Dirichlet boundary

To demonstrate that also a more complex Dirichlet boundary condition can be reconstructed, the previous example is repeated with a synthetic Dirichlet function η_D^{exact} that consists of the composition of two sinusoidal functions with different periods (figure 6.10b):

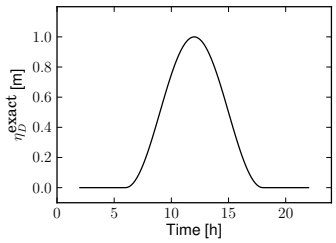
$$\eta_D^{\text{exact}}(t) = \begin{cases} 0 & \text{if } t < 6 \text{ h}, \\ \frac{1}{2}\left(\cos\left(\frac{2\pi(t-6\text{ h})}{p}\right) - \cos\left(\frac{8\pi(t-6\text{ h})}{p}\right)\right) & \text{if } 6 \text{ h} \leq t \leq 18 \text{ h}, \\ 0 & t > 18 \text{ h}. \end{cases}$$

where $p \coloneqq 12$ h. The smoothing value is obtained from the guideline equation (6.4) and is $\alpha = 0.43$. The observations are generated in the same way as in the previous example and are plotted in figure 6.10a.

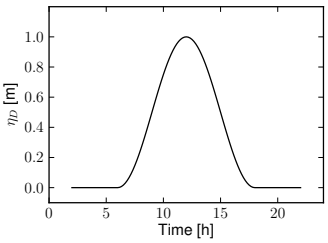
In this case, the optimisation tolerance was not reached after 150 optimisation iterations (158 functional evaluations), at which point the algorithm was manually terminated. The results in figure 6.10 show a reconstruction of the Dirichlet boundary condition up to an absolute error of 0.011 m, which corresponds to an error of approximately 0.5% of the incoming wave height. Comparing the convergence plot 6.10e to the previous examples shows that the shape of the wave profile to be reconstructed can have significant impact on the convergence of the optimisation method.



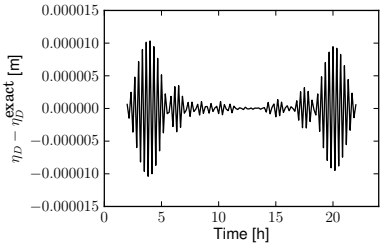
(a) The wet/dry interface observations constructed by running the forward problem with the synthetic Dirichlet boundary values. The observations are approximated indicator functions of the wet/dry interface (marked as dashed line).



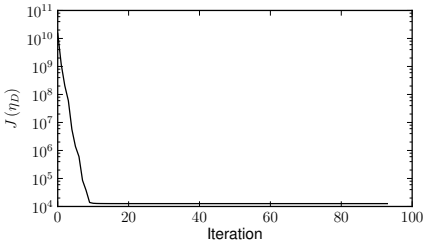
(b) Synthetic Dirichlet boundary values.



(c) Reconstructed Dirichlet boundary values.

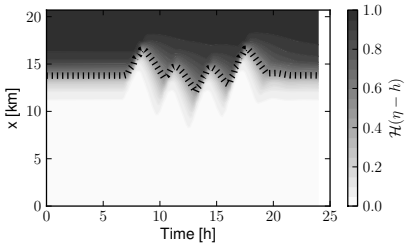


(d) Difference between synthetic and reconstructed Dirichlet boundary values.

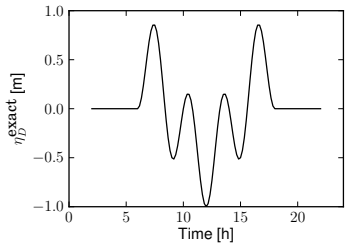


(e) Functional values during the optimisation iterations.

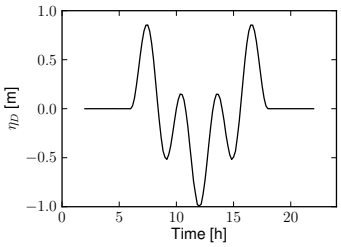
Figure 6.9.: Results of the wave profile reconstruction on a sloping beach with a sinusoidal Dirichlet boundary and smoothing value $\alpha = 0.43$. The initial and final 2 h of the boundary values are excluded from the reconstruction.



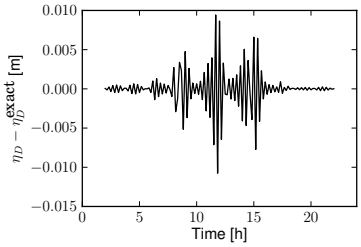
(a) The wet/dry interface observations constructed by running the forward problem with the synthetic Dirichlet boundary values. The observations are approximated indicator functions of the wet/dry interface (marked as dashed line).



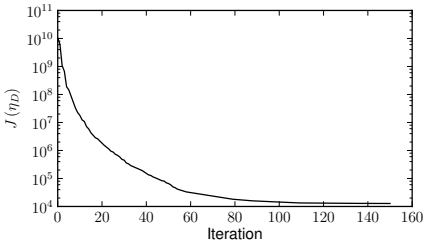
(b) Synthetic Dirichlet boundary values.



(c) Reconstructed Dirichlet boundary values.

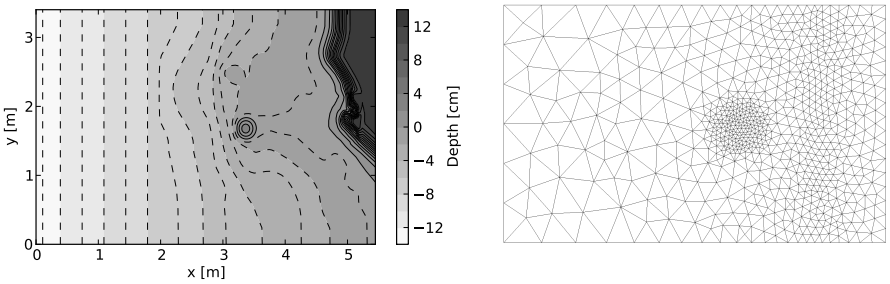


(d) Difference between synthetic and reconstructed Dirichlet boundary values.



(e) Functional values during the optimisation iterations.

Figure 6.10.: Results of the wave profile reconstruction on a sloping beach with a composed sinusoidal Dirichlet boundary and smoothing value $\alpha = 0.43$. The initial and final 2 h of the boundary values are excluded from the reconstruction.



(a) Bathymetry (b) Mesh

Figure 6.11.: The laboratory setup of the Hokkaido-Nansei-Oki tsunami example, based on 1/400 laboratory experiment by the Central Research Institute for Electric Power Industry. The island at the center and the coast on the right are hit by a tsunami shaped wave coming from the left boundary.

6.4.2. Reconstruction of the Hokkaido-Nansei-Oki tsunami wave profile

The second application is motivated by the question of wether it is possible to reconstruct a tsunami wave profile from satellite observations that record the inundation line over time.

The considered event is the Hokkaido-Nansei-Oki tsunami that occurred in 1993 and produced run-up heights of up to 30 m on the Okushiri island, Japan. The Central Research Institute for Electric Power Industry (CRIEPI) in Abiko, Japan constructed a 1/400 laboratory model of the area around the island (Matsuyama and Tanaka, 2001). Following Yalciner et al. (2011), this experiment is simulated in a rectangular domain of size 5.448 m $\times$ 3.402 m. The bathymetry and coastal topography is shown in figure 6.11a. It contains an island in the center and coastal regions on the top right of the domain. On the left boundary a surface elevation profile is enforced that resembles a tsunami wave (figure 6.12b). The task is to reconstruct this wave profile. On the remaining boundaries, a no-normal flow condition is imposed.

The domain is discretised with an unstructured mesh consisting of $1,411$ triangular elements with increasing resolution near the inundation areas, see figure 6.11b. The temporal discretisation uses a time step of 0.5 s with a total simulation time of 32 s. The Manning coefficient is set to $\mu = 0.05$ s/m$^{\frac{1}{3}}$. The observations are synthetically generated by running the model with the

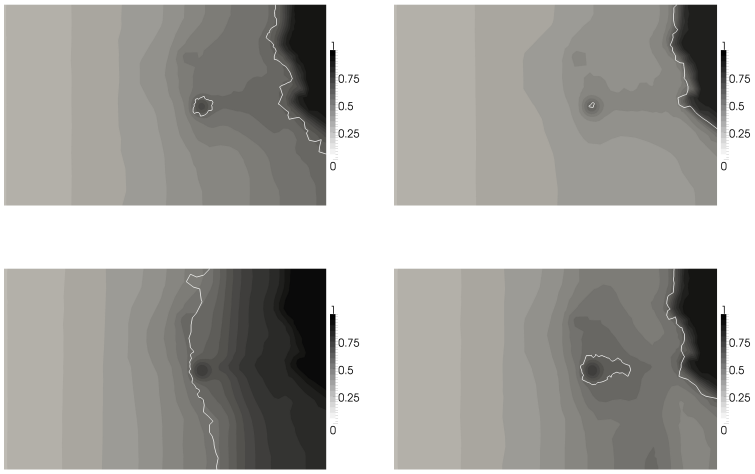
wave profile that was used in the laboratory experiment while recording the wet/dry interface. The smoothness value was determined experimentally: with a value of $\alpha = 0.05$, the Newton solver in the forward model failed to converge during the optimisation procedure. For this reason, an α value of 0.1 was used.

As an initial guess for the optimisation algorithm, the Dirichlet boundary values are set to $1.05 \cdot 10^{-3}$ m for all time levels, which corresponds to the final free-surface displacement of the input wave. For the same reason as in the examples above, the initial and final 2 s of the Dirichlet boundary values were then reset to the correct Dirichlet boundary values and excluded from the optimisation. Furthermore, a box constraint was used to restrict the minimum and maximum free-surface displacement to -1.5 cm and $+2$ cm; without these constraints the first optimisation iterations would generate unrealistically large Dirichlet boundary values for which the forward model would not converge.

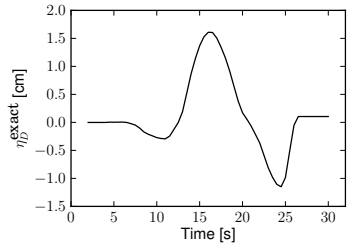
After 103 optimisation iterations (113 functional evaluations) the relative decrease of the functional of interest per iteration fell below the machine precision and the algorithm terminated. The results are shown in figure 6.12. The incoming wave was reconstructed up to an absolute error of $3.91 \cdot 10^{-7}$ cm (figure 6.12d), which corresponds to an relative error of less than $3 \cdot 10^{-5}\%$ of the incoming wave height.

6.5. Summary

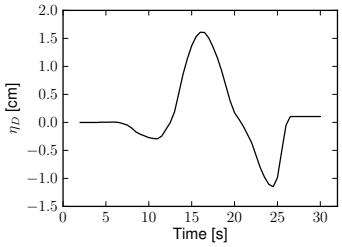
A shallow water model with wetting and drying and its adjoint model has been developed and used in the context of data assimilation. The formulated optimisation problem reconstructs the wave profile of inundation problems, such that the difference between observed and simulated wet/dry interface is minimised. The examples demonstrate that, under idealised conditions, the profile of the incoming wave can be recovered with an error of order 1% or less of the wave height, even if no or very little information is known a priori. This is the first step towards reconstructing unknowns such as the tsunami source and wave profile from real inundation data that is available from satellite imaging.



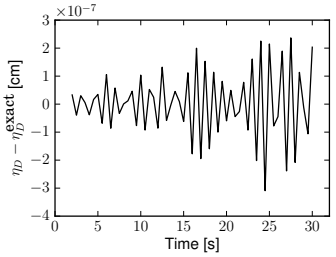
(a) The wet/dry interface observations after 0 s, 20 s, 29 s, 32 s, in reading order. The observations are constructed by running the forward problem with the synthetic Dirichlet boundary values. The observations are approximated indicator functions of the wet/dry interface (marked as white lines)



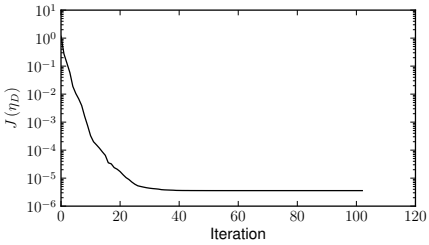
(b) The correct tsunami wave profile.



(c) The reconstructed tsunami wave profile.



(d) The difference between correct and reconstructed tsunami wave profiles.



(e) The functional values during the optimisation iterations

Figure 6.12.: Results of the reconstruction of the Hokkaido-Nansei-Oki tsunami wave profile. The initial and final 2 s of the boundary values are excluded from the reconstruction.

Chapter 7

Summary and future work

7.1. Summary

The beginning of this thesis introduced PDE-constrained optimisation and the concept of solving the adjoint equations to efficiently compute derivative information. This chapter specifically focused on the required software development, in particular of the adjoint model, and possible automation.

Chapter 2 presented the library libadjoint that facilitates the development of adjoint models based on the abstraction that the computer model is a sequence of equation solves. To demonstrate the utility of this library, it was used to adjoin two models in the computational fluid dynamics framework Fluidity, for which the application of current low-level algorithmic differentiation tools is impossible.

Chapter 3 applied this library to the finite-element environment FEniCS. By combining the high-level abstractions that libadjoint and FEniCS provide, it was possible to fully automate the derivation and implementation of adjoint models from a given forward model. Multiple examples demonstrated the degree of automation and the robustness of this approach.

Chapter 4 used these techniques to develop a framework for solving optimisation problems governed by PDEs. Its key features are its generality by relying on the FEniCS environment, the small amount of development effort required due to the automation with dolfin-adjoint and its flexibility by interfacing with a wide range of gradient-free and gradient-based optimisation algorithms.

This framework was then applied in chapter 5 to solve for the optimal positioning of tidal turbines in a tidal stream. The results showed an increase in power extraction of up to 75% compared to a reference layout and resulted in novel turbine configurations.

Another application of the optimisation module was presented in chapter 6, where data assimilation was applied to a shallow water model with wetting and drying. The question addressed here was: is it possible to reconstruct the profile of an incoming wave from observations of the inundation pattern? An idealised tsunami inundation was investigated, for which the tsunami wave profile was successfully recovered.

7.2. Future work

7.2.1. Application of libadjoint to other high-level frameworks

The library libadjoint (chapter 2) was specifically designed to develop adjoint models of existing forward models. In this thesis, this was demonstrated by applying libadjoint to parts of Fluidity and to the FEniCS system. The high-level approaches of FEniCS and libadjoint enabled the automated derivation and solution of adjoint systems. Future work could apply libadjoint to additional forward models, with a focus on models where the problem description is accessible as data, in order to achieve a similar degree of automation. An example is the framework proposed by Markall et al. (2010) and Rathgeber et al. (2012), a related project to FEniCS that aims to create optimised parallel finite-element code for multiple target platforms, including multi-core CPUs and GPUs from UFL. However, the derivation of adjoint models is useful outside the context of finite element methods, and the concepts and ideas of libadjoint apply to any computer model that can be naturally represented as a sequence of equation solves.

7.2.2. Automated continuous adjoint approach in FEniCS

Chapters 2 and 3 focused on the automatic derivation and solution of the discrete adjoint approach. However, as described in section 1.3.1, there are cases where the continuous adjoint approach is preferable, for example in

combination with adaptive discretisation methods, discontinuous numerical schemes or shape optimisation.

The automation of the continuous adjoint approach from low-level source code is impossible, since the forward equations are only available in their discretised form, interwoven with machine-specific implementation details.

In contrast, in FEniCS the user specifies the discretised weak form in the high-level form language UFL, and hence the mathematical problem specification is directly accessible as data. The continuous weak problem is obtained by replacing the discrete functions with functions from suitable continuous function spaces (and by removing numerical facilitators such as upwinding). Assuming that FEniCS also had an abstraction layer for the temporal discretisation, the same strategy could be used to obtain the continuous formulation of time-dependent problems.

With the continuous weak formulation of the forward system, the associated continuous weak formulation of the adjoint formulation could be derived automatically by symbolic manipulation. The result could then be discretised and solved with the techniques that FEniCS provides.

7.2.3. Using the automated adjoint derivation and the PDE-constrained optimisation framework

The results of chapters 2 and 3 enable the rapid development of forward and adjoint models. These can then be directly used with the optimisation framework presented in chapter 4 to solve PDE-constrained optimisation problems. The ease of setting up these problems is a major advance: it enables the development of complex PDE-constrained optimisation in days instead of years. Extensive documentation, examples, and precompiled packages for both FEniCS and dolfin-adjoint further simplify the application of these tools.

A major avenue is to take this work and translate it into impact across of science and engineering. So far, dolfin-adjoint has been used to solve adjoint models and PDE-constrained optimisation problems for the Burgers', the Yamabe, and the shallow ice equation, the shallow water equations with wetting and drying, the Navier-Stokes, the Cahn-Hilliard and the Gray-Scott equations and more. Therefore, future work could consist

of developing additional forward and adjoint models and their application in PDE-constrained optimisation algorithms.

7.2.4. Support for multilevel methods for the PDE-constrained optimisation framework

Multilevel optimisation methods solve the optimisation problem on a hierarchy of approximations to accelerate its solution (Dreyer et al., 2000; Lewis and Nash, 2005). The approximation levels can either be defined manually, for example by generating multiple meshes with different resolutions, or be determined automatically, for example with reduced order techniques (Fahl and Sachs, 2003) or error estimators (Ziems and Ulbrich, 2011).

The advantage of multilevel methods is that often most optimisation iterations can be performed on the coarsest approximation level. This is particularly important in the case of PDE-constrained optimisation, in which the computational cost is typically dominated by the PDE-solves. This cost can be greatly reduced if a coarser discretisation is used. Therefore, future work could extend the optimisation framework presented in chapter 4 to support multilevel optimisation.

7.2.5. Support for solving optimisation problems in the unreduced form

Currently, the optimisation module of dolfin-adjoint solves the optimisation problem in the reduced form (4.3). However, for the reasons explained in section 4.2, it might be advantageous to solve the problem in the unreduced form. The unreduced formulation leads to a large-scale optimisation problem and requires specially designed optimisation algorithms. Such algorithms are implemented in the software package TAO (Munson et al., 2012), and hence future work could consist of interfacing the presented optimisation framework with TAO.

7.2.6. Extensions of the optimal positioning of tidal turbines

The layout optimisation for tidal turbine farms described in chapter 5 can be extended in various ways.

To begin with, the results presented in the chapter are possibly only local solutions to the optimisation problem, since a local gradient-based opti-

misation algorithm was applied. Therefore, the next step could consist of applying a global (or hybrid) optimisation algorithm to investigate possible better configurations. Alternatively, one could perform multiple gradient-based optimisations each with a different initial turbine layout to explore more of the solution space.

Secondly, the example in section 5.6.3 suggested that reducing the number of turbines could lead to a better cost-performance ratio. Such questions could be addressed by applying mixed-integer programming, which enables the optimisation of both continuous variables such as the turbine positions, as well as discrete variables such as the numbers of turbines.

Thirdly, the optimisation formulation could be made more realistic by using a functional of interest that includes factors such as the installation costs of the cables or the dependency of the installation cost on the water depth. In addition, time-dependent control variables could be used, for example for controlling the blade pitches or the turbine directions over time.

Finally, the forward model could be made more realistic by adding a depth-averaged turbulence model, such as the $k-\epsilon$ model proposed by Rastogi and Rodi (1978). Furthermore, the turbine model should be validated against measurements, or compared against three-dimensional simulations.

7.2.7. Adjoint model of the Navier-Stokes equations with free-surface and wetting and drying

The data assimilation presented in chapter 6 used the shallow water equations with wetting and drying to simulate the fluid dynamics. However, in the inundation areas, the vertical velocity component typically becomes significant, and non-hydrostatic effects important. Depending on the problem at hand it can therefore be questioned whether the depth-averaged hydrostatic shallow-water equations sufficiently resolve these processes.

The author has published a wetting and drying algorithm for the non-hydrostatic Navier-Stokes equations with free-surface (Funke et al., 2011). In this approach, the three-dimensional domain adapts vertically with the movement of the free-surface. The wetting and drying algorithm stops this movement below a certain threshold to avoid negative water heights, which naturally leads to a no-normal flow boundary condition on the free-surface boundary.

Future work could develop an adjoint model for this wetting and drying algorithm. Besides a maximum operator which can be smoothed with similar techniques to these in chapter 6, the scheme does not contain any non-differentiable operators and is therefore suitable for the adjoint model derivation.

Appendix A

Adjoint derivation of the continuous Burgers equation

Consider the Burger's equation with Dirichlet boundary conditions:

$$\begin{aligned} \frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u = f \quad & \text{in } \Omega \times (0,T), \\ u = g \quad & \text{on } \partial\Omega \times (0,T), \\ u = u_0 \quad & \text{for } \Omega \times \{0\}, \end{aligned} \tag{A.1}$$

where $\Omega \times (0,T)$ defines the domain of interest, $u : \Omega \times (0,T) \to \mathbb{R}^d$ is the solution, $\nu \in \mathbb{R}$ is the viscosity parameter, $f : \Omega \times (0,T) \to \mathbb{R}^d$ is the source function, $g : \partial\Omega \times (0,T) \to \mathbb{R}^d$ describes the boundary value and $u_0 : \Omega \to \mathbb{R}^d$ the initial value. The functional of interest J is assumed to be to be a time-space integral.

Equations (A.1) can be written in the canonical form $F(u) = 0$ by defining

$$F(u) \coloneqq \begin{pmatrix} \frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u - f \\ u|_{\partial\Omega \times (0,T)} - g \\ u|_{\Omega \times \{0\}} - u_0 \end{pmatrix}. \tag{A.2}$$

In the canonical form the adjoint equation reads as:

$$\frac{\partial F}{\partial u}^* \lambda = \frac{\partial J}{\partial u}^*, \tag{A.3}$$

where λ is the adjoint solution. Taking the derivative of operator (A.2) in direction v yields

$$\frac{\partial F}{\partial u} v = \begin{pmatrix} \frac{\partial v}{\partial t} + (v \cdot \nabla)\, u + (u \cdot \nabla) v - \nu \nabla^2 v \\ v|_{\partial\Omega,(0,T)} \\ v|_{\Omega \times \{0\}} \end{pmatrix}. \tag{A.4}$$

To determine the adjoint version of this operator, the weak formulation of equation (A.3) is used: find λ such that all v:

$$\left\langle \frac{\partial F}{\partial u}^* \lambda, v \right\rangle = \left\langle \lambda, \frac{\partial F}{\partial u} v \right\rangle = \left\langle \frac{\partial J}{\partial u}^*, v \right\rangle.$$

Substituting equation (A.4) and the functional of interst into this weak formulation yields: find $\lambda \coloneqq (\lambda_1, \lambda_2, \lambda_3)^*$ such that for all v:

$$\begin{aligned} &\left\langle \lambda_1, \frac{\partial v}{\partial t} + (v \cdot \nabla)\, u + (u \cdot \nabla) v - \nu \nabla^2 v \right\rangle_{\Omega \times (0,T)} + \\ &\langle \lambda_2, v \rangle_{\partial\Omega \times (0,T)} + \langle \lambda_3, v \rangle_{\Omega \times \{0\}} = \left\langle \frac{\partial J}{\partial u}^*, v \right\rangle_{\Omega \times (0,T)}. \end{aligned} \tag{A.5}$$

In order to obtain the strong formulation of the adjoint equation, the terms in equation (A.5) are integrated by parts such that the test function v is separated to one side of the inner products. Careful integration by parts yields: find $(\lambda_1, \lambda_2, \lambda_3)$ such that for all v:

$$\begin{aligned} &\left\langle -\frac{\partial \lambda_1}{\partial t} + (\nabla u)^* \lambda_1 - (\nabla \cdot u)\, \lambda_1 - u \cdot \nabla \lambda_1 - \nu \nabla^2 \lambda_1, v \right\rangle_{\Omega \times (0,T)} \\ &\left\langle (u \cdot n) \lambda_1 + \lambda_2 + \nu \frac{\partial \lambda_1}{\partial n}, v \right\rangle_{\partial\Omega \times (0,T)} - \left\langle \nu \lambda_1, \frac{\partial v}{\partial n} \right\rangle_{\partial\Omega \times (0,T)} + \\ &+ \langle \lambda_1, v \rangle_{\Omega \times \{T\}} + \langle \lambda_3 - \lambda_1, v \rangle_{\Omega \times \{0\}} = \left\langle \frac{\partial J}{\partial u}^*, v \right\rangle_{\Omega \times (0,T)}. \end{aligned} \tag{A.6}$$

Since the test function v is arbitrary, the strong form of the Burgers' adjoint equation is obtained by sorting the terms:

$$
\begin{aligned}
-\frac{\partial \lambda_1}{\partial t} + (\nabla u)^* \lambda_1 - (\nabla \cdot u)\,\lambda_1 - (u \cdot \nabla)\,\lambda_1 - \nu \nabla^2 \lambda_1 &= \frac{\partial J^*}{\partial u} && \text{in } \Omega \times (0,T), \\
\lambda_1 &= 0 && \text{on } \partial\Omega \times (0,T), \\
\lambda_1 &= 0 && \text{for } \Omega \times \{T\}, \\
-\nu \frac{\partial \lambda_1}{\partial n} - (u \cdot n)\lambda_1 &= \lambda_2 && \text{on } \partial\Omega \times (0,T), \\
\lambda_1 &= \lambda_3 && \text{at } \Omega \times \{0\}.
\end{aligned}
$$

Note that adjoint components λ_2 and λ_3 appear only in the last two equations. Thus, unless their values are of interest, these equations may be ignored.

Appendix B

Example usage of algorithmic differentiation

In this section, an algorithmic differentiation tool is applied to compute the derivatives in forward and reverse mode for the function:

$$f(m) = \sqrt{\sin(m)^2}, \quad m \in \mathbb{R}. \tag{B.1}$$

The following code implements this function:

```
double f(double m)
{
  double f1, f2, f3;
  f1 = sin(m);
  f2 = f1 * f1;
  f3 = sqrt(f2);
  return f3;
}
```

Equation (B.1) can be rewritten as:

$$f_3 \circ f_2 \circ f_1(m), \tag{B.2}$$

with:

$$f_3(\cdot) = \sqrt{\cdot}, \quad f_2(\cdot) = \cdot^2, \quad f_1(\cdot) = \sin(\cdot).$$

Let m_i denote the intermediate values of this equation, for example $m_2 = f_2(f_1(m))$. Then the derivative is obtained by applying the chain rule to equation (B.2):

$$\frac{df}{dm} = f_3'(m_2)f_2'(m_1)f_1'(m), \tag{B.3}$$

where:

$$f_3'(m_2) = \frac{1}{2\sqrt{m_2}}, \quad f_2'(m_1) = 2m_1, \quad f_1'(m) = \cos(m).$$

The forward mode of an AD tool produces code that computes the directional derivative $\frac{df}{dm}m_d$ by evaluating equation (B.3) from the right to the left along with the evaluation of the function itself. The source to source AD tool TAPENADE (Hascoët and Pascual, 2004) generates following code in forward mode:

```
/*
  Differentiation of f in forward (tangent) mode:
   variations of useful results: f
   with respect to varying inputs: m
   RW status of diff variables: f:out m:in
*/
double f_d(double m, double md, double *f) {
    double f1, f2, f3;
    double f1d, f2d, f3d;
    f1d = md*cos(m);
    f1 = sin(m);
    f2d = f1d*f1 + f1*f1d;
    f2 = f1*f1;
    f3d = (f2 == 0.0 ? 0.0 : f2d/(2.0*sqrt(f2)));
    f3 = sqrt(f2);
    *f = f3;
    return f3d;
}
```

The reverse mode of an AD tool generates code that first evaluates the function itself and stores all intermediate values and then computes $\left(\frac{df}{dm}\right)^* f_b$ by evaluation equation (B.3) from the left to the right. The application of TAPENADE in reverse mode returns:

```
/*
  Differentiation of f in reverse (adjoint) mode:
   gradient of useful results: f m
   with respect to varying inputs: m
   RW status of diff variables: f:in-killed m:incr
*/
void f_b(double m, double *mb, double fb) {
    double f1, f2, f3;
    double f1b, f2b, f3b;
    double f;
    f1 = sin(m);
    f2 = f1*f1;
    f3b = fb;
    if (f2 == 0.0)
        f2b = 0.0;
    else
        f2b = f3b/(2.0*sqrt(f2));
    f1b = 2*f1*f2b;
    *mb = *mb + cos(m)*f1b;
}
```

Appendix C

Adjoint derivation of the shallow water equations

Consider the shallow water equations with appropriate boundary and initial conditions:

$$\begin{aligned} \frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u + g \nabla \eta = C\|u\|u, \\ \frac{\partial \eta}{\partial t} + \nabla \cdot (Hu) = 0, \end{aligned} \tag{C.1}$$

where $\Omega \times (0,T)$ denotes the domain of interest, $u : \Omega \times (0,T) \to \mathbb{R}^d$ and $\eta : \Omega \times (0,T) \to \mathbb{R}$ are the unknown depth-averaged velocity and free-surface displacement, respectively, $H \in \mathbb{R}$ is the basin depth, $g \in \mathbb{R}$ is the gravitational force, $\nu \in \mathbb{R}$ is the viscosity coefficient and $C : \Omega \to \mathbb{R}$ specifies the friction. The functional of interest is denoted as $J(u) \in \mathbb{R}$.

Equations (C.1) can be written in the canonical form $F(u,\eta) = 0$ by defining

$$F(u,\eta) \coloneqq \begin{pmatrix} \frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u + g \nabla \eta - C\|u\|u \\ \frac{\partial \eta}{\partial t} + \nabla \cdot (Hu) \end{pmatrix}. \tag{C.2}$$

In the canonical form the adjoint equation reads as:

$$\frac{\partial F}{\partial z}^* \lambda = \frac{\partial J}{\partial z}^*, \tag{C.3}$$

where $z \coloneqq (u, \eta)$ and λ is the adjoint solution. Taking the derivative of operator (C.2) in direction $v \coloneqq (v_u, v_\eta)$ yields

$$\frac{\partial F}{\partial z} v = \begin{pmatrix} \frac{\partial v_u}{\partial t} + v_u \cdot \nabla u + u \cdot \nabla v_u - \nu \nabla^2 v_u - C\left(\|u\| v_u + \frac{u}{\|u\|} u \cdot v_u \right) + g \nabla v_\eta, \\ \frac{\partial v_\eta}{\partial t} + \nabla \cdot (H v_u) \end{pmatrix}. \tag{C.4}$$

To determine the adjoint version of this operator, the weak formulation of equation (C.3) is used: find λ such that all v:

$$\left\langle \frac{\partial F}{\partial z}^* \lambda, v \right\rangle = \left\langle \lambda, \frac{\partial F}{\partial z} v \right\rangle = \left\langle \frac{\partial J}{\partial z}^* , v \right\rangle. \tag{C.5}$$

Substituting equation (C.4) and the functional of interest into equation (C.5) yields the weak adjoint formulation: find $\lambda \coloneqq (\lambda_u, \lambda_\eta)^*$ such that for all v:

$$\begin{aligned} &\left\langle \lambda_u, \frac{\partial v_u}{\partial t} + v_u \cdot \nabla u + u \cdot \nabla v_u - \nu \nabla^2 v_u - C\left(\|u\| v_u + \frac{u}{\|u\|} u \cdot v_u \right) + g \nabla v_\eta \right\rangle_{\Omega \times (0,T)} \\ &\quad + \left\langle \lambda_\eta, \frac{\partial v_\eta}{\partial t} + \nabla \cdot (H v_u) \right\rangle_{\Omega \times (0,T)} = \left\langle \frac{\partial J}{\partial u}^* , v_u \right\rangle_{\Omega \times (0,T)} + \left\langle \frac{\partial J}{\partial \eta}^* , v_\eta \right\rangle_{\Omega \times (0,T)}. \end{aligned} \tag{C.6}$$

In order to obtain the strong adjoint formulation the terms in equation (C.6) are integrated by parts such that the test function v is separated to one side of the inner products. For brevity the following derivation does not show the boundary integrals. Careful integration by parts yields: find $(\lambda_u, \lambda_\eta)$ such that for all v:

$$\begin{aligned} &\left\langle -\frac{\partial \lambda_u}{\partial t} + (\nabla u)^* \lambda_u - (\nabla \cdot u) \lambda_u - u \cdot \nabla \lambda_u - \nu \nabla^2 \lambda_u - H \nabla \lambda_\eta, v_u \right\rangle_{\Omega \times (0,T)} \\ &- C \left\langle \|u\| \lambda_u + \frac{u \cdot \lambda_u}{\|u\|} u, v_u \right\rangle_{\Omega \times (0,T)} - \left\langle \frac{\partial \lambda_\eta}{\partial t} + g \nabla \cdot \lambda_u, v_\eta \right\rangle_{\Omega \times (0,T)} \\ &= \left\langle \frac{\partial J}{\partial u}^* , v_u \right\rangle_{\Omega \times (0,T)} + \left\langle \frac{\partial J}{\partial \eta}^* , v_\eta \right\rangle_{\Omega \times (0,T)}. \end{aligned}$$

Since the test function v is arbitrary, the strong form of the adjoint shallow water equations is obtained by sorting the terms:

$$
\begin{aligned}
-\frac{\partial \lambda_u}{\partial t} + (\nabla u)^* \lambda_u - (\nabla \cdot u)\, \lambda_u - u \cdot \nabla \lambda_u - \nu \nabla^2 \lambda_u & \\
-H \nabla \lambda_\eta - C \left(\|u\| \lambda_u + \frac{u \cdot \lambda_u}{\|u\|} u \right) &= \frac{\partial J^*}{\partial u} \,, \\
-\frac{\partial \lambda_\eta}{\partial t} - g \nabla \cdot \lambda_u &= \frac{\partial J^*}{\partial \eta} \,.
\end{aligned}
$$

References

Alnæs, M. S. 2011. UFL: A finite element form language. In *Automated Solution of Differential Equations by the Finite Element Method*, A. Logg, K. A. Mardal, and G. N. Wells, Eds. Springer-Verlag, Berlin, Heidelberg, New York, Chapter 17, 299–334. 72, 73, 74, 78

Anderson, W. K. and Venkatakrishnan, V. 1999. Aerodynamic design optimization on unstructured grids with a continuous adjoint formulation. *Computers & Fluids 28,* 4–5, 443–480. 15

Andersson, J., Åkesson, J., and Diehl, M. 2012. CasADi – a symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation*, S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, Eds. Lecture Notes in Computational Science and Engineering. Springer-Verlag, Berlin, Heidelberg, New York. 27

Ascher, U. M. and Petzold, L. R. 1998. *Computer methods for ordinary differential equations and differential-algebraic equations.* Vol. 61. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 12

Ascher, U. M., Ruuth, S. J., and Spiteri, R. J. 1997. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics 25,* 2-3, 151–167. 151, 152

Bäck, T. 1996. *Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, Oxford, UK. 121

Bagchi, A. and Brummelhuis, P. T. 1994. Parameter identification in tidal models with uncertain boundaries. *Automatica 30,* 5, 745–759. 145

Bagheri, B. and Scott, R. 2004. About Analysa. Tech. rep. 16

Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. 2010. PETSc users manual. Tech. Rep. ANL-95/11, Argonne National Laboratory, USA. Revision 3.1. 28, 44

Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. 1997. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, Basel, Switzerland, 163–202. 28, 44

Balzano, A. 1998. Evaluation of methods for numerical simulation of wetting and drying in shallow water flow models. *Coastal Engineering 34,* 1-2, 83–107. 153, 159

Bangerth, W., Hartmann, R., and Kanschat, G. 2007. deal.II - a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software 33,* 4, 24. 16

Becker, R., Meidner, D., and Vexler, B. 2005. RODOBO: A C++ library for optimization with stationary and nonstationary PDEs. http://www.rodobo.uni-hd.de. 93

Ben Elghali, S. E., Balme, R., Le Saux, K., El Hachemi Benbouzid, M., Charpentier, J. F., and Hauville, F. 2007. A simulation model for the evaluation of the electrical power potential harnessed by a marine current turbine. *IEEE Journal of Oceanic Engineering 32,* 4, 786–797. 122

Bernard, P. E., Remacle, J. F., Comblen, R., Legat, V., and Hillewaert, K. 2009. High-order discontinuous Galerkin schemes on general 2D manifolds applied to the shallow water equations. *Journal of Computational Physics 228,* 17, 6514–6535. 63

Biegler, L. T., Ghattas, O., Heinkenschloss, M., and van Bloemen Waanders, B. 2003. Large-scale PDE-constrained optimization: An introduction. In *Large-scale PDE-constrained optimization*, L. T. Biegler, O. Ghattas, M. Heinkenschloss, and B. Bloemen Waanders, Eds. Springer-Verlag, Berlin, Heidelberg, New York, 3–13. 121

Bilbao, M. and Alba, E. 2009. Simulated annealing for optimization of wind farm annual profit. In *2nd International Symposium on Logistics and Industrial Informatics, 2009*. 1–5. 120, 122

Bischof, C. H., Bücker, H. M., Rasch, A., Slusanschi, E., and Lang, B. 2007. Automatic differentiation of the general-purpose computational fluid dynamics package FLUENT. *Journal of Fluids Engineering 129,* 5, 652–658. 18

Blaise, S., St-Cyr, A., Mavriplis, D., and Lockwood, B. 2012. Discontinuous Galerkin unsteady discrete adjoint method for real-time efficient tsunami simulations. *Journal of Computational Physics*. in press. 145

Boggs, P. T. and Tolle, J. W. 1995. Sequential quadratic programming. *Acta Numerica 4,* 1, 1–51. 100

Broyden, C. G. 1970. The convergence of a class of double-rank minimization algorithms 1. General considerations. *IMA Journal of Applied Mathematics 6,* 1, 76–90. 98

Broyden, C. G., Dennis Jr, J. E., and More, J. J. 1973. On the local and superlinear convergence of quasi-Newton methods. *IMA Journal of Applied Mathematics 12,* 3, 223–245. 98

Bryden, I. G. and Couch, S. J. 2007. How much energy can be extracted from moving water with a free surface: A question of importance in the field of tidal current energy? *Renewable Energy 32,* 11, 1961–1966. 119

Bücker, H. M., Rasch, A., and Wolf, A. 2004. A class of OpenMP applications involving nested parallelism. In *Proceedings of the 19th ACM Symposium on Applied Computing – SAC 2004*. ACM Press, New York, USA, 220–224. 28

Bücker, M., Corliss, G., Hovland, P., Naumann, U., and Norris, B. 2006. *Automatic differentiation: Applications, theory and implementations*. Lecture Notes in Computational Science and Engineering Series, vol. 50. Springer-Verlag, Berlin, Heidelberg, New York. 18

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. 1995. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing 16,* 5, 1190–1208. 65, 99

Chen, X. and Navon, I. M. 2009. Optimal control of a finite-element limited-area shallow-water equations model. *Studies in Informatics and Control 18,* 1, 41–62. 145

Cioaca, A., Alexe, M., and Sandu, A. 2012. Second-order adjoints for solving PDE-constrained optimization problems. *Optimization Methods and Software 27,* 4-5, 625–653. 97

Comblen, R., Lambrechts, J., Remacle, J. F., and Legat, V. 2010. Practical evaluation of five partly discontinuous finite element pairs for the non-conservative shallow water equations. *International Journal for Numerical Methods in Fluids 63,* 6, 701–724. 148

Cotter, C. J. and Ham, D. A. 2011. Numerical wave propagation for the triangular P1DG-P2 finite element pair. *Journal of Computational Physics 230,* 8, 2806–2820. 67

Cotter, C. J., Ham, D. A., and Pain, C. C. 2009. A mixed discontinuous/continuous finite element pair for shallow-water ocean modelling. *Ocean Modelling 26,* 1-2, 86–90. 63, 67, 148

Crank, J. and Nicolson, P. 1947. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Mathematical Proceedings of the Cambridge Philosophical Society 43,* 1, 50–67. 60, 67

Dennis, Jr., J. E. and Moré, J. J. 1977. Quasi-Newton methods, motivation and theory. *SIAM Review 19*, 46–89. 97

Ding, Y. and Wang, S. 2006. Optimal control of open-channel flow using adjoint sensitivity analysis. *Journal of Hydraulic Engineering 132,* 11, 1215–1228. 145

Ding, Y. and Wang, S. S. Y. 2005. Identification of Manning's roughness coefficients in channel network using adjoint analysis. *International Journal of Computational Fluid Dynamics 19,* 1, 3–13. 145

Divett, T., Vennell, R., and Stevens, C. 2011. Optimisation of multiple turbine arrays in a channel with tidally reversing flow by numerical modelling with adaptive mesh. In *Proceedings of the European Wave and Tidal Energy Conference 2011*. Southampton, UK. 119, 122, 123, 124

Draper, S., Houlsby, G. T., Oldfield, M. L. G., and Borthwick, A. G. L. 2010. Modelling tidal energy extraction in a depth-averaged coastal domain. *Renewable Power Generation, IET 4,* 6, 545–554. 133

Dreyer, T., Maar, B., and Schulz, V. 2000. Multigrid optimization in applications. *Journal of Computational and Applied Mathematics 120,* 1–2, 67–84. 172

Dular, P., Geuzaine, C., Henrotte, F., and Legros, W. 1998. A general environment for the treatment of discrete problems and its application to the finite element method. *IEEE Transactions on Magnetics 34,* 5, 3395–3398. 16

Elman, H. C., Silvester, D. J., and Wathen, A. J. 2005. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Oxford University Press, USA. 12

Errico, R. M. 1997. What is an adjoint model? *Bulletin of the American Meteorological Society 78,* 11, 2577–2591. 3, 121

Fagerfjäll, P. 2010. Optimizing wind farm layout- more bang for the buck using mixed integer linear programming. M.S. thesis, Chalmers University of Technology, Gothenburg, Sweden. 122

Fahl, M. and Sachs, E. W. 2003. Reduced order modelling approaches to PDE-constrained optimization based on proper orthogonal decomposition. In *Large-Scale PDE-Constrained Optimization*, L. T. Biegler, M. Heinkenschloss, O. Ghattas, B. Bloemen Waanders, T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, and T. Schlick,

Eds. Vol. 30. Springer-Verlag, Berlin, Heidelberg, New-York, 268–280. 172

Fang, F., Piggott, M. D., Pain, C. C., Gorman, G. J., and Goddard, A. J. H. 2006. An adaptive mesh adjoint data assimilation method. *Ocean Modelling 15,* 1-2, 39–55. 14

Farrell, P. E. and Funke, S. W. 2011. *libadjoint: A library for developing adjoint models (Version 0.9.2)*. http://launchpad.net/libadjoint. 38, 44, 50

Farrell, P. E., Ham, D. A., Funke, S. W., and Rognes, M. E. 2012. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*. accepted. 5, 82

Fletcher, R. 1970. A new approach to variable metric algorithms. *The Computer Journal 13,* 3, 317–322. 98

Fletcher, R. 1980. *Practical Methods of Optimization, Vol. 1: Unconstrained Optimization.* John Wiley & Sons, Chichester, UK. 99

Förster, M., Naumann, U., and Utke, J. 2011. Toward adjoint OpenMP. Tech. Rep. AIB-2011-13, RWTH Aachen, Germany. 19, 26

Foster, I. 1995. *Designing and building parallel programs.* Vol. 95. Addison-Wesley, Reading, MA, USA. 28

Funke, S. W., Pain, C. C., Kramer, S. C., and Piggott, M. D. 2011. A wetting and drying algorithm with a combined pressure/free-surface formulation for non-hydrostatic models. *Advances in Water Resources 34,* 11, 1483–1495. 5, 145, 173

Garrett, C. and Cummins, P. 2008. Limits to tidal current power. *Renewable Energy 33,* 11, 2485–2490. 119

Gejadze, I. Y. and Copeland, G. J. M. 2005. Adjoint sensitivity analysis for fluid flow with free surface. *International Journal for Numerical Methods in Fluids 47,* 8-9, 1027–1034. 145

Geuzaine, C. and Remacle, J.-F. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering 79,* 11, 1309–1331. 154

Ghil, M. and Malanotte-Rizzoli, P. 1991. Data assimilation in meteorology and oceanography. In *Advances in Geophysics*, R. Dmowska and B. Saltzman, Eds. Vol. 33. Elsevier, 141–266. 2

Giering, R. and Kaminski, T. 2002. Recomputations in reverse mode AD. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, G. Corliss, A. Griewank, C. Faurè, L. Hascoët, and U. Naumann, Eds. Automatic Differentiation of Algorithms: From Simulation to Optimization. Springer-Verlag, Berlin, Heidelberg, New York, Chapter 33, 283–291. 53

Giering, R. and Kaminski, T. 2003. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. *Proceedings in Applied Mathematics and Mechanics 2,* 1, 54–57. 18

Gilbert, J. C. 1992. Automatic differentiation and iterative processes. *Optimization Methods and Software 1,* 1, 13–21. 76

Giles, M. B. and Pierce, N. A. 2000. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion 65,* 3-4, 393–415. 92, 128

Gill, P. E. and Murray, W. 1972. Quasi-Newton methods for unconstrained optimization. *IMA Journal of Applied Mathematics 9,* 1, 91–108. 99

Goldfarb, D. 1970. A family of variable-metric methods derived by variational means. *Mathematics of Computation 24,* 109, 23–26. 98

Gourgue, O. 2011. Finite element modeling of sediment dynamics in the Scheldt. Ph.D. thesis, École polytechnique de Louvain, Belgium. 156

Gray, P. and Scott, S. K. 1983. Autocatalytic reactions in the isothermal, continuous stirred tank reactor: Isolas and other forms of multistability. *Chemical Engineering Science 38,* 1, 29–43. 86

Griesse, R. and Walther, A. 2004. Evaluating gradients in optimal control: Continuous adjoints versus automatic differentiation. *Journal of Optimization Theory and Applications 1221,* 1, 63–86. 15, 22, 129

Griewank, A. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software 1,* 1, 35–54. 52, 53, 54, 56

Griewank, A. 2003. A mathematical view of automatic differentiation. *Acta Numerica 12*, 321–398. 18

Griewank, A. and Walther, A. 2000. Algorithm 799: revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software 26,* 1, 19–45. 53

Griewank, A. and Walther, A. 2008. *Evaluating derivatives: Principles and techniques of algorithmic differentiation* Second Ed. SIAM, Philadelphia, PA, USA. 11, 17, 18

Grimm, J., Pottier, L., and Rostaing-Schmidt, N. 1996. Optimal time and minimum space-time product for reversing a certain class of programs. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, PA, USA, 95–106. 52, 56

Guillaume, P. and Masmoudi, M. 1994. Computation of high order derivatives in optimal shape design. *Numerische Mathematik 67,* 2, 231–250. 11, 97

Gunzburger, M. D. 2003. *Perspectives in Flow Control and Optimization.* Advances in Design and Control. SIAM, Philadelphia, PA, USA. 15, 51, 128, 129

Hascoët, L. and Araya-Polo, M. 2006. Enabling user-driven checkpointing strategies in reverse-mode automatic differentiation. In *Proceedings of the ECCOMAS CFD 2006 conference*, E. O. P. Wesseling and J. Périaux, Eds. INRIA, Egmond aan Zee, Netherlands, 19. 19

Hascoët, L. and Pascual, V. 2004. Tapenade 2.1 user's guide. Tech. Rep. RT-0300, INRIA Sophia Antipolis, Sophia Antipolis, France. 17, 18, 60, 180

Heimbach, P., Hill, C., and Giering, R. 2005. An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation. *Future Generation Computer Systems 21,* 8, 1356–1371. 18, 52

Heimbach, P., Menemenlis, D., Losch, M., Campin, J. M., and Hill, C. 2010. On the formulation of sea-ice models. Part 2: Lessons from multi-year adjoint sea-ice export sensitivities through the Canadian Arctic Archipelago. *Ocean Modelling 33,* 1-2, 145–158. 19

Hesthaven, J. S. and Warburton, T. 2008. *Nodal discontinuous Galerkin methods: Algorithms, analysis, and applications.* Vol. 54. Springer-Verlag, Berlin, Heidelberg, New York. 148

Hinze, M. and Kunisch, K. 2001. Second order methods for optimal control of time-dependent fluid flow. *SIAM Journal on Control and Optimization 40,* 3, 925–946. 12

Hinze, M., Pinnau, R., Ulbrich, M., and Ulbrich, S. 2009. *Optimization with PDE constraints.* Mathematical Modelling: Theory and Applications Series, vol. 23. Springer-Verlag, Berlin, Heidelberg, New York. 7, 11, 95, 96

Hock, W. and Schittkowski, K. 1983. A comparative performance evaluation of 27 nonlinear programming codes. *Computing 30,* 4, 335–358. 100

Hovland, P. D. 1997. Automatic differentiation of parallel programs. Ph.D. thesis, University of Illinois, Champaign, USA. 19, 26

Huang, H.-S. 2009. Efficient hybrid distributed genetic algorithms for wind turbine positioning in large wind farms. In *IEEE International Symposium on Industrial Electronics 2009.* 2196–2201. 121, 122

Jameson, A. 1988. Aerodynamic design via control theory. *Journal of Scientific Computing 3,* 3, 233–260. 2

Jones, E., Oliphant, T., Peterson, P., et al. 2001. SciPy: Open source scientific tools for Python. 65, 102, 127, 158

Kärnä, T., de Brye, B., Gourgue, O., Lambrechts, J., Comblen, R., Legat, V., and Deleersnijder, E. 2011. A fully implicit wetting–drying method for DG-FEM shallow water models, with an application to the Scheldt estuary. *Computer Methods in Applied Mechanics and Engineering 200,* 5–8, 509 – 524. 145, 147, 148, 150, 151, 153, 155, 156, 160

Kawahara, M. and Kawasaki, T. 1990. A flood control of dam reservoir by conjugate gradient and finite element methods. In *Proceedings of the Fifth International Conference on Numerical Ship Hydrodynamics*, K. Mori, Ed. The National Academies Press, Washington, D.C., USA, 45–56. 145

Kennedy, J. and Eberhart, R. 1995. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks, 1995*. Vol. 4. 1942–1948. 121

Kim, J., Hunke, E., and Lipscomb, W. 2006. A sensitivity-enhanced simulation approach for community climate system model. In *Proceedings of the 6th international conference on Computational Science – ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Springer-Verlag, Berlin, Heidelberg, New York, 533–540. 18

Kirby, R. C. and Logg, A. 2006. A compiler for variational forms. *ACM Transactions on Mathematical. Software 32,* 3, 417–444. 74

Kowalik, Z., Knight, W., Logan, T., and Whitmore, P. 2005. Numerical modeling of the global tsunami: Indonesian tsunami of 26 December 2004. *Science of Tsunami Hazards 23,* 1, 40– 56. 145

Kowarz, A. and Walther, A. 2006. Optimal checkpointing for time-stepping procedures in ADOL-C. In *Proceedings of the 6th international conference on Computational Science – ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Vol. 3994. Springer-Verlag, Berlin, Heidelberg, New York, 541–549. 19, 53

Kraft, D. 1988. A software package for sequential quadratic programming. *Forschungsbericht / Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt 88,* 28. 127

Laarhoven, P. J. M. and Aarts, E. H. L., Eds. 1987. *Simulated annealing: Theory and applications.* Kluwer Academic Publishers, Norwell, MA, USA. 102, 121

Lackner, M. A. and Elkinton, C. N. 2007. An analytical framework for offshore wind farm layout optimization. *Wind Engineering 31,* 1, 17–31. 122

Le Dimet, F. X. and Talagrand, O. 1986. Variational algorithms for analysis and assimilation of meteorological observations: Theoretical aspects. *Tellus A 38A,* 2, 97–110. 92

Lebigot, E. 2011. *Uncertainties: A Python module for calculations with uncertainties (Version 1.5.4).* 60

Lee, S. H., Lee, S. H., Jang, K., Lee, J., and Hur, N. 2010. A numerical study for the optimal arrangement of ocean current turbine generators in the ocean current power parks. *Current Applied Physics 10,* 2, Supplement, S137 – S141. 119

Lewis, R. M. and Nash, S. G. 2005. Model problems for the multigrid optimization of systems governed by differential equations. *SIAM J. Sci. Comput. 26,* 6, 1811–1837. 172

Lions, J. L. 1971. *Optimal control of systems governed by partial differential equations.* Vol. 27. Springer-Verlag, Berlin, Heidelberg, New York. 92

Liu, D. C. and Nocedal, J. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical Programming 45,* 1, 503–528. 99

Liu, Z. and Sandu, A. 2005. Analysis of discrete adjoints for upwind numerical schemes. In *Proceedings of the 5th international conference on Computational Science – ICCS 2005*, V. Sunderam, G. van Albada, P. Sloot, and J. Dongarra, Eds. Springer-Verlag, Berlin, Heidelberg, New York, 7–13. 15, 81

Logg, A. 2007. Automating the finite element method. *Archives of Computational Methods in Engineering 14,* 2, 93–138. 28, 72

Logg, A., Mardal, K.-A., Wells, G. N., et al. 2011. *Automated Solution of Differential Equations By the Finite Element Method.* Springer-Verlag, Berlin, Heidelberg, New-York. 16, 28, 72, 74, 81, 152

Logg, A. and Wells, G. N. 2010. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software 37,* 2. 72

Logg, A., Wells, G. N., and Hake, J. 2011. DOLFIN: A C++/Python finite element library. In *Automated Solution of Differential Equations by the Finite Element Method*, A. Logg, K. A. Mardal, and G. N. Wells, Eds. Springer-Verlag, Berlin, Heidelberg, New-York, Chapter 10. 72

Long, K., Boggs, P. T., and van Bloemen Waanders, B. G. 2012. Sundance: High-level software for PDE-constrained optimization. *Scientific Programming*. Preprint. 16, 73

Lyness, J. N. and Moler, C. B. 1967. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis 4,* 2, 202–210. 11

Markall, G. R., Ham, D. A., and Kelly, P. H. J. 2010. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science 1,* 1, 1815–1823. 170

Martins, J. R. R. A., Sturdza, P., and Alonso, J. J. 2003. The complex-step derivative approximation. *ACM Transactions on Mathematical Software 29,* 3, 245–262. 11

Matsuyama, M. and Tanaka, H. 2001. An experimental study of the highest runup height in the 1993 Hokkaido Nansei-Oki earthquake tsunami. In *National Tsunami Hazard Mitigation Program Review and International Tsunami Symposium (ITS 2001).* 879–889. 165

Medeiros, S. C. and Hagen, S. C. 2012. Review of wetting and drying algorithms for numerical tidal flow models. *International Journal for Numerical Methods in Fluids*. Preprint. 145, 146

Menemenlis, D., Campin, J., Heimbach, P., Hill, C., Lee, T., Nguyen, A., Schodlock, M., and Zhang, H. 2008. ECCO2: High resolution global ocean and sea ice data synthesis. *Mercator Ocean Quarterly Newsletter 31*, 13–21. 2, 121

Meza, J. C. 1994. OPT++: An object-oriented class library for nonlinear optimization. Tech. Rep. SAND94-8225, Sandia National Laboratory, California, USA. 104

Mikkelsen, R. 2003. Actuator disc methods applied to wind turbines. Ph.D. thesis, Technical University of Denmark, Denmark. 122

Miyaoka, T. and Kawahara, M. 2008. Optimal control of drainage basin considering moving boundary. *International Journal of Computational Fluid Dynamics 22,* 10, 677–686. 146

Munson, T., Sarich, J., Wild, S., Benson, S., and McInnes, L. C. 2012. TAO 2.1 users manual. Tech. Rep. ANL/MCS-TM-322, Argonne National Laboratory, USA. 104, 172

Nardi, L., Sorror, C., Badran, F., and Thiria, S. 2009. YAO: A software for variational data assimilation using numerical models. In *Proceedings of the International Conference on Computational Science and Its Applications – ICCSA 2009, Seoul, Korea, 2009*, O. Gervasi, D. Taniar, B. Murgante, A. Laganà, Y. Mun, and M. L. Gavrilova, Eds. Springer-Verlag, Berlin, Heidelberg, New-York, 621–636. 27

Naumann, U. 2012. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation.* Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, USA. 4, 19

Navon, I. M. 1998. Practical and theoretical aspects of adjoint parameter estimation and identifiability in meteorology and oceanography. *Dynamics of Atmospheres and Oceans 27,* 1–4, 55–79. 92

Nelder, J. A. and Mead, R. 1965. A simplex method for function minimization. *The Computer Journal 7,* 4, 308–313. 102

Nocedal, J. and Wright, S. J. 1999. *Numerical optimization* 2 Ed. Springer-Verlag, Berlin, Heidelberg, New-York. 3, 11, 96, 97, 100, 102

Oliphant, T. E. 2006. *A Guide to NumPy*. Vol. 1. Trelgol Publishing, USA. 42, 44

Palacios, F., Alonso, J., Colonno, M., Hicken, J., and Lukaczyk, T. 2012. Adjoint-based method for supersonic aircraft design using equivalent area distributions. In *50th AIAA Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, Inc., Nashville, Tennessee, USA. 93

Pape, A. L. and Lecanu, J. 2004. 3D Navier-Stokes computations of a stall-regulated wind turbine. *Wind Energy 7,* 4, 309–324. 122

Park, S. K. and Xu, L. 2009. *Data Assimilation for Atmospheric, Oceanic and Hydrologic Applications*. Springer-Verlag, Berlin, Heidelberg, New-York. 64

Pearson, J. E. 1993. Complex patterns in a simple system. *Science 261,* 5118, 189–192. 86, 87

Pelletier, D., Turgeon, E., Lacasse, D., and Borggaard, J. 2003. Adaptivity, sensitivity, and uncertainty: Toward standards of good practice in computational fluid dynamics. *AIAA journal 41,* 10, 1925–1933. 14

Piggott, M. D., Gorman, G. J., Pain, C. C., Allison, P. A., Candy, A. S., Martin, B. T., and Wells, M. R. 2008. A new computational framework for multi-scale ocean modelling based on adapting unstructured meshes. *International Journal for Numerical Methods in Fluids 56,* 8, 1003–1015. 28

Powell, M. J. D. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal 7,* 2, 155–162. 102

Powell, M. J. D. 1994. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Proceedings of the Sixth Workshop on optimization and numerical analysis*, S. Gomez and J. P. Hennart, Eds. Oaxaca, Mexico, 51–67. 102

Powell, M. J. D., Skolnick, M. M., and Tong, S. S. 1991. Interdigitation: A hybrid technique for engineering design optimization employing genetic algorithms, expert systems, and numerical optimization. *Handbook of Genetic Algorithms*, 312–331. 126

Raffard, R. L. and Tomlin, C. J. 2005. Second order adjoint-based optimization of ordinary and partial differential equations with application to air traffic flow. In *Proceedings of the 2005 American Control Conference*. 798–803. 12, 97

Rall, L. B. and Corliss, G. F. 1996. An introduction to automatic differentiation. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, PA, USA, 1–17. 18

Rastogi, A. K. and Rodi, W. 1978. Predictions of heat and mass transfer in open channels. *Journal of the Hydraulics Division 104,* 3, 397–420. 173

Rathgeber, F., Markall, G. R., Mitchell, L., Loriant, N., Ham, D. A., Bertolli, C., and Kelly, P. H. J. 2012. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC) 2012*. Salt Lake City, Utah, USA. to appear. 170

Renders, J. M. and Flasse, S. P. 1996. Hybrid methods using genetic algorithms for global optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B 26,* 2, 243–258. 126

Roache, P. J. 2002. Code verification by the method of manufactured solutions. *Journal of Fluids Engineering 124,* 1, 4–10. 130

Rudolph, G. 1996. Convergence of evolutionary algorithms in general search spaces. In *Proceedings of IEEE International Conference on Evolutionary Computation*. IEEE Press, Piscataway, New Jersey, USA, 50–54. 120

Salari, K. and Knupp, P. 2000. Code verification by the method of manufactured solutions. Tech. Rep. SAND2000-1444, Sandia National Laboratories, Albuquerque, NM, USA. 111, 130

Salcedo-Sanz, S., Saavedra-Moreno, B., Paniagua-Tineo, A., Prieto, L., and Portilla-Figueras, A. 2011. A review of recent evolutionary computation-based techniques in wind turbines layout optimization problems. *Central European Journal of Computer Science 1*, 101–107. 121

Salter, S. H. 2009. Correcting the under-estimate of the tidal-stream resource of the Pentland firth. In *Proceedings of the 8th European Wave and Tidal Energy Conference – EWTEC 2009*, M. Leijon and M. Berg, Eds. Uppsala, Sweden. 141

Samizo, Y. and Kawahara, M. 2011. Optimal control of shallow water flows using adjoint equation method. *Advanced Materials Research 403-408*, 466–469. 145

Sanders, B. F. and Katopodes, N. D. 1996. Optimal control of sudden water release from a reservoir. In *Managing Water: Coping with Scarcity and Abundance*, M. English and A. Szollosi-Nagy, Eds. San Francisco, USA, 314–319. 145

Sanders, B. F. and Katopodes, N. D. 2000. Adjoint sensitivity analysis for shallow-water wave control. *Journal of Engineering Mechanics 126,* 9, 909–919. 145

Sandu, A. 2006. On the properties of Runge-Kutta discrete adjoints. In *Proceedings of the 6th international conference on Computational Science – ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Springer-Verlag, Berlin, Heidelberg, New-York, 550–557. 21

Schiesser, W. E. 1991. *The numerical method of lines: Integration of partial differential equations*. Vol. 29. Academic Press, San Diego, USA. 12

Seeger, M. 2008. Low rank updates for the Cholesky decomposition. Tech. rep., University of California at Berkeley, Berkeley, California, USA. 99

Shanno, D. F. 1970. Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation 24,* 111, 647–656. 98

Sirkes, Z. and Tziperman, E. 1997. Finite difference of adjoint or adjoint of finite difference? *Monthly Weather Review 125,* 12, 3373–3378. 17, 21

Song, L., Zhou, J., Li, Q., Yang, X., and Zhang, Y. 2011. An unstructured finite volume model for dam-break floods with wet/dry fronts over complex topography. *International Journal for Numerical Methods in Fluids 67,* 8, 960–980. 145

Stammer, D., the ECCO Consortium, Chassignet, E., and the HYCOM Constortium. 2000. Ocean state estimation and prediction in support of oceanographic research. *Oceanography 2,* 13, 51–56. 2

Stammer, D. and Wunsch, C. 1999. Computational requirements for ECCO in support of CLIVAR and GODAE. The ECCO Report Series 2, Scripps Institution of Oceanography, Massachusetts Institute of Technology, Boston, USA. 3

Stein, W. A. et al. 2011. *Sage Mathematics Software (Version 5.3).* 15

Stumm, P. and Walther, A. 2009. Multistage approaches for optimal offline checkpointing. *SIAM Journal on Scientific Computing 31,* 3, 1946–1967. 56

Stumm, P. and Walther, A. 2010. New algorithms for optimal online checkpointing. *SIAM Journal on Scientific Computing 32,* 2, 836–854. 56

Taylor, C. and Hood, P. 1973. A numerical solution of the Navier-Stokes equations using the finite element technique. *Computers & Fluids 1,* 1, 73–100. 128

Tber, M. H., Hascoët, L., Vidard, A., and Dauvergne, B. 2007. Building the tangent and adjoint codes of the ocean general circulation model OPA with the automatic differentiation tool TAPENADE. Tech. Rep. RR-6372, INRIA Sophia Antipolis, Sophia Antipolis, France. 52

Thacker, W. C. 1981. Some exact solutions to the nonlinear shallow-water wave equations. *Journal of Fluid Mechanics 107*, 499–508. 152, 153

Thevenin, D. and Janiga, G. 2008. *Optimization and computational fluid dynamics*. Springer-Verlag, Berlin, Heidelberg, New-York. 11

Tröltzsch, F. 2005. *Optimale Steuerung partieller Differentialgleichungen - Theorie, Verfahren und Anwendungen* Zweite Auflage Ed. Vieweg+Teubner, Vieweg, Wiesbaden, Germany. 95, 105, 113

Utke, J., Hascoet, L., Heimbach, P., Hill, C., Hovland, P., and Naumann, U. 2009. Toward adjoinable MPI. In *IEEE International Symposium on Parallel Distributed Processing, 2009.* IEEE, Rome, Italy, 1–8. 19, 26, 29

Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., and Wunsch, C. 2008. OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software 34,* 4, 1–36. 18

van Bloemen Waanders, B., Bartlett, R., Long, K., Boggs, P., and Salinger, A. 2002. Large scale non-linear programming for PDE constrained optimization. Tech. Rep. SAND2002-3198, Sandia National Laboratories, New Mexico, California, USA. 93

van Rossum, G., Drake, F. L., et al. 2008. Python reference manual. 28, 59

Vennell, R. 2010. Tuning turbines in a tidal channel. *Journal of Fluid Mechanics 663*, 253–267. 119

Vennell, R. 2011. Tuning tidal turbines in-concert to maximise farm efficiency. *Journal of Fluid Mechanics 671,* 1, 587–604. 119, 134

Vennell, R. 2012. The energetics of large tidal turbine arrays. *Renewable Energy 48,* 0, 210–219. 125, 132, 134

Vermeer, L. J., Sørensen, J. N., and Crespo, A. 2003. Wind turbine wake aerodynamics. *Progress in Aerospace Sciences 39,* 6-7, 467–510. 122

Vidard, A., Vigilant, F., Deltel, C., and Benshila, R. 2008. *NEMO tangent & adjoint models (NemoTam): Reference manual & user's guide.* Paris, France. ANR-08-COSI-016. 19, 51

Walther, A. and Griewank, A. 2004. Advantages of binomial checkpointing for memory-reduced adjoint calculations. In *Proceedings of ENUMATH 2003*, M. Feistauer, V. Dolejí, P. Knobloch, and K. Najzar, Eds. Springer-Verlag, Berling, Heidelberg, New-York, 834–843. 52

Wan, C., Wang, J., Yang, G., and Zhang, X. 2010. Optimal micro-siting of wind farms by particle swarm optimization. In *Advances in Swarm Intelligence*, Y. Tan, Y. Shi, and K. Tan, Eds. Lecture Notes in Computer Science Series, vol. 6145. Springer-Verlag, Berlin, Heidelberg, New-York, 198–205. 121

Wang, Y., Yagola, A. G., and Yang, C. 2010. *Optimization and regularization for computational inverse problems and applications.* Springer-Verlag, Berlin, Heidelberg, New-York. 96, 162

Westerink, J. J., Luettich, R. A., Feyen, J. C., Atkinson, J. H., Dawson, C., Roberts, H. J., Powell, M. D., Dunion, J. P., Kubatko, E. J., and Pourtaheri, H. 2008. A basin- to channel-scale unstructured grid hurricane storm surge model applied to southern Louisiana. *Monthly Weather Review 136,* 3, 833–864. 145

Xiao, Q., Kuo, Y. H., Ma, Z., Huang, W., Huang, X. Y., Zhang, X., Barker, D. M., Michalakes, J., and Dudhia, J. 2008. Application of an adiabatic WRF adjoint to the investigation of the May 2004 McMurdo, Antarctica, severe wind event. *Monthly Weather Review 136,* 10, 3696–3713. 18

Xue, H. and Du, Y. 2010. Implementation of a wetting-and-drying model in simulating the Kennebec–Androscoggin plume and the circulation in Casco Bay. *Ocean Dynamics 60,* 2, 341–357. 145

Yalciner, A. C., Imamura, F., and Synolakis, C. E. 2011. Amplitude evolution and runup of long waves: Comparison of experimental and numerical data on a 3D complex topography. In *Advanced Numerical Models For Simulating Tsunami Waves And Runup*, P. L. Liu, H. Yeh, and C. Synolakis, Eds. World Scientific Publishing Company, Chapter 9, 243–247. 165

Zhang, J., Lin, J., Sun, X., and Chen, C. 2009. Application of an improved wetting and drying scheme in POM. In *Proceedings of the 2009 International Conference on Energy and Environment Technology - Volume 02 – ICEET 2009*. IEEE Computer Society, Washington, DC, USA, 427–432. 145

Ziems, J. and Ulbrich, S. 2011. Adaptive multilevel inexact SQP methods for PDE-constrained optimization. *SIAM Journal on Optimization 21,* 1, 1–40. 172