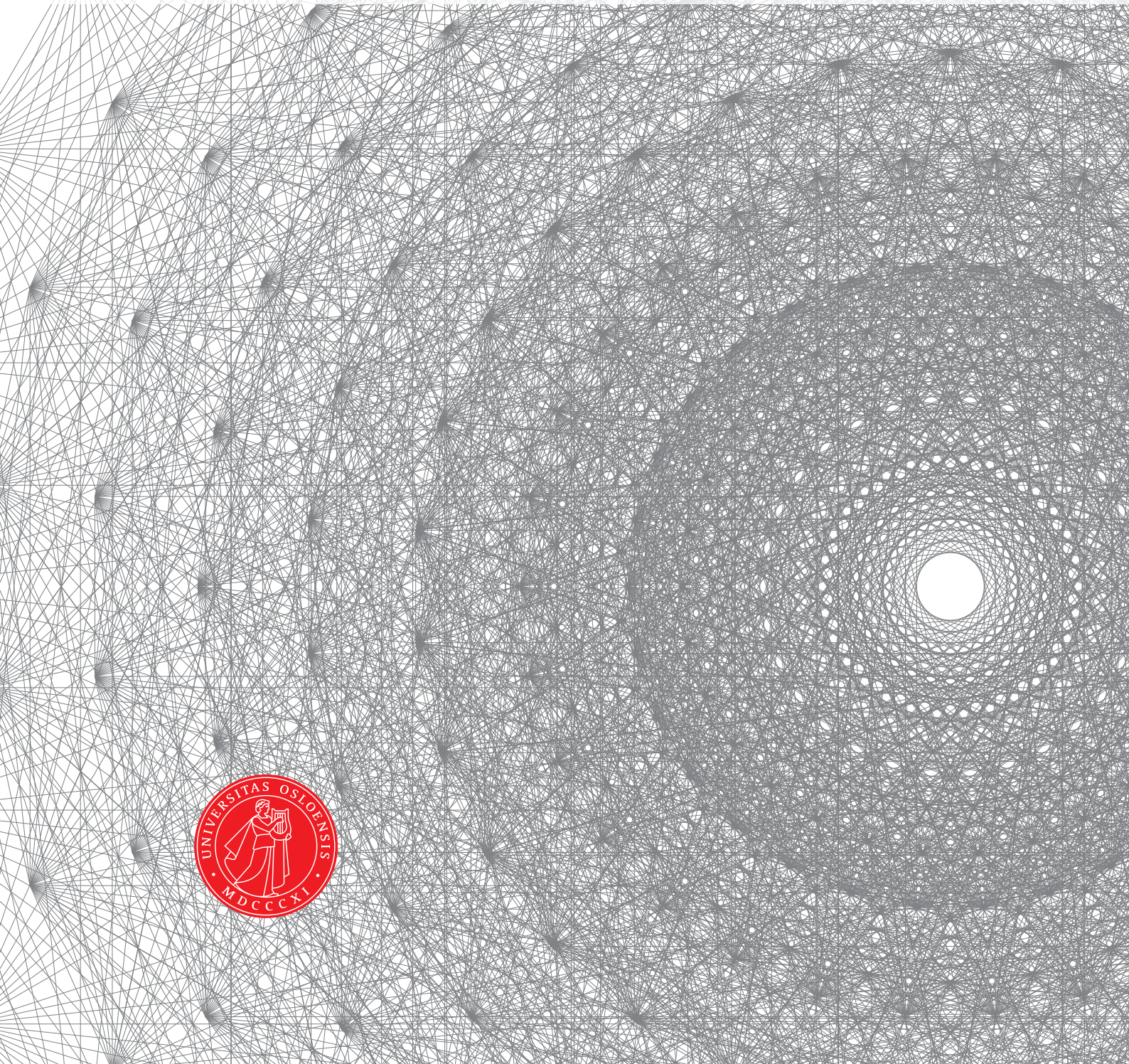
UiO : Department of Mathematics
University of Oslo

An Algorithmic Differentiation Tool for FEniCS

Sebastian Kenji Mitusch

Master's Thesis, Spring 2018



This master's thesis is submitted under the master's programme *Computational Science and Engineering*, with programme option *Computational Science*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Acknowledgements

Firstly, I would like to thank my supervisor Simon Wolfgang Funke, for giving me the opportunity to work on this project. Your valuable input and positive attitude made this project possible. I would also like to thank my secondary supervisor Kent-Andre Mardal, I am especially grateful for your feedback during the end of this project.

Furthermore, I would like to thank David Ham and Lawrence Mitchell for the collaboration on developing support for Firedrake. Particularly, thank you David for your comments on Dirichlet boudary controls. Also, thank you to everyone who contributed to the pyadjoint repository. Thank you, Jørgen Dokken for providing helpful comments and tips for the thesis.

I would like to thank Lars Magnus Valnes, Geir Ringstad, and Per Kristian Eide for providing me with data for the brain diffusion model. Thank you to my friends and family, who supported me throughout.

Contents

Chapter 1

Introduction

Optimization problems seek to find the input that minimizes (or maximizes) a function from a set of available inputs. These kinds of problems can occur in many different fields. In science and engineering, optimization problems frequently appear in combination with partial differential equations (PDEs), as these equations can be used to describe physical processes. These problems can be formulated as

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

where $J(u, m) \in \mathbb{R}$ is some objective functional, U and M are suitable Banach spaces, and $F(u, m) = 0$ is one or multiple PDEs parametrized by m with solution u. Problems of the type (1.1) are called PDE-constrained optimization problems. Algorithms for solving these types of problems can be divided into two classes: gradient-free algorithms and gradient-based algorithms. Although gradient-free algorithms can be very useful, they tend to scale poorly with the dimension of the parameter space M [44]. Therefore our focus will be on gradient-based algorithms. In the context of gradient-based optimization algorithms, a particular challenge is efficiently computing derivatives of J.

Since there is an implicit relationship between u and m given by the PDE constraint, it can be useful to reformulate (1.1). Assuming that any $m \in M$ yields a unique $u \in U$, one can define the reduced functional as $\hat{J}(m) = J(u(m), m)$. Then the problem becomes $\min_{m \in M} \hat{J}(m)$. To solve this using gradient-based optimization algorithms, we need to find the derivative of $\hat{J}(m) = J(u(m), m)$ with respect to m.

A common numerical method for computing derivatives is the finite difference method. However, a tricky part of this method is the choice of a step size. If the step size is too big, the approximations will be poor because of large truncation errors. On the other hand, if the step size is too small the result can be dominated by numerical imprecision [22]. Even without worrying about step size, computing finite differences can be computationally expensive. Gradient-based optimization algorithms need a full gradient of $\hat{J}$, but a single finite difference computation only gives a directional derivative. Thus, for each iteration of a gradient-based algorithm, we need to perform $\dim(M) + 1$ PDE solves. Combining the fact that problems of this type often have a parameter space with $dim(M) \gg 1$ and that solving the PDE comes at a significant com-

putational cost, finite differences is typically an infeasible option for practical problems of type (1.1).

Another option is finding the derivative of $\hat{J}(m)$ by the tangent linear approach. This involves the solution of a linearised version of the constraint PDE to obtain the directional derivative. There is no need for any step size, and we do not get any truncation errors either. As the tangent linear model is not an approximation, the source of error stems only from discretization and numerical precision. However, the tangent linear method still does not solve the problem of the high computational expense. We must still compute the tangent linear model for each basis function in the parameter space, resulting in $dim(M)$ linear PDE solves to obtain the gradient.

For most PDE-constrained optimization problems, the adjoint method is the most efficient choice. From a single evaluation of a linear PDE, the adjoint equation, whose computational cost is comparable to the original model, the full gradient is determined. The adjoint method has a higher memory cost, but in comparison to the tangent linear and finite difference methods, the adjoint method is the superior choice when the dimensions of the parameter space is larger than 2. For this reason the adjoint approach is commonly used in engineering. Jameson pioneered the use of adjoint methods in aeronautical engineering in [23]. Here he considered how to optimize the shape of a aerofoil, using the inviscid Euler equations for compressible flow as governing equations. In recent years, a discrete adjoint approach based on algorithmic differentiation was developed for Reynolds-averaged Navier-Stokes equations [33]. Gradient-based optimization using this to obtain gradients was shown to have a high level of efficiency and robustness for aerodynamic problems relating to optimal design of aerofoils [34]. The adjoint of Reynolds-averaged Navier-Stokes equations can also be used for optimal positioning of wind turbines in wind farms [26]. We call these problems design problems, as they try to find parameters which optimize a design.

The adjoint model has two main implementation approaches. The implementation of the *continuous adjoint*, and the implementation of the *discrete adjoint*. In the continuous approach the adjoint equations are derived from the continuous model, prior to discretization. After this, the adjoint PDE can be discretized and implemented using any suitable technique. In addition, the physical meaning of the boundary condition is more clear in the continuous approach [13]. On the other hand, with a continuous method the derivation and discretization of the adjoint model must be completed manually.

The discrete adjoint approach, instead derives the adjoint model from the discretized model. This is illustarted in figure 1.1. The discrete approach yields an exact gradient of the discrete objective functional [13]. That is, the computed gradient is consistent with the discrete implementation of the forward PDE. Given that the forward PDE has been discretized and implemented, one can use algorithmic differentiation (AD) to obtain an adjoint model automatically. AD works by decomposing the implementation into elementary instructions (like plus, minus, multiplication) and differentiating by applying the chain rule. More details on AD is presented in section 2.2. Although AD is in principal a straightforward method, difficulty arises when applying it to existing simulation software. In order to efficiently integrate AD implementations, expert understanding of the software and model is required [38, page xii]. This stems from the fact that a basic AD tool cannot separate implementation

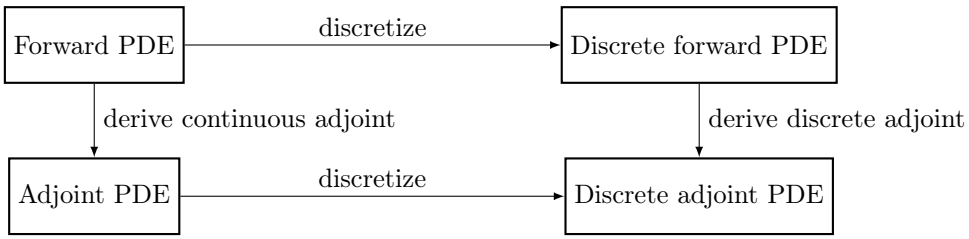


Figure 1.1: The two different approaches for deriving the adjoint model. The discrete approach performs the discretize step before deriving the discrete adjoint PDE. Note that the two approaches for deriving adjoints do not necessarily commute.

details and mathematics. Instead, black-box AD tools have to deal with low-level implementation details such as parallel communication and I/O. Despite this, low-level AD has been successfully applied to large simulation software such as OpenFOAM [42] and MITgcm [18].

To maintain more of a high level mathematical abstraction, the software package dolfin-adjoint [10] is implemented as an AD tool exploiting the high level representation of the discrete model in the domain-specific language UFL [3]. In UFL the discrete variational formulation of a PDE can be implemented with near mathematical notation. The variational formulations written in UFL are represented symbolically, which enables efficient manipulation of the model. In particular, one can efficiently differentiate these representations with respect to parameters. This approach has proven to be very efficient [10] and dolfin-adjoint has become a popular package for deriving discrete adjoint models. However, there are currently some limitations in the dolfin-adjoint implementation. Because the package was implemented with high level AD in mind, it assumes that such a high level abstraction is available. This limits the scope of the AD tool to only certain environments. A particular problem that has arisen from this limitation is that objective functionals are restricted to the classes of functionals that dolfin-adjoint is familiar with. Furthermore, dolfin-adjoint does not support strong Dirichlet boundary conditions as controls, and instead relies on weakly imposing these boundary conditions through the Nitsche method [5].

In this thesis we will focus on implementing a generic AD python package and apply it to the finite element framework FEniCS [2]. The resulting implementation will then derive the discrete adjoint models and serve as a successor to dolfin-adjoint. The aim is to support both strong implementations of Dirichlet boundary condition controls, in addition to enabling AD for arithmetic operations where a high level abstraction is not available.

AD is not exclusively used in programs aimed at solving PDEs. It has also been widely adopted for machine learning, and especially neural networks. Torch [8] and TensorFlow [1] are examples of machine learning software employing AD. The new implementation of dolfin-adjoint could allow for experimentation on mixing PDE and machine learning.

At the time of writing, some results from this thesis has been presented at conferences. The dolfin-adjoint framework was presented as a poster at the FEniCS'17 conference, receiving an award for the best poster. Some of the

work presented in chapter 6 was part of a talk by Simon W. Funke during the FEniCS'18 conference.

1.1 Structure

The thesis is structured as follows. In chapter 2 we present necessary and useful background on topics that will be extensively used in the thesis. Chapter 3 introduces a generic AD software, and describes how it is implemented. Next, chapter 4 details how we apply this generic AD tool to a finite-element framework. Examples of applying the resulting AD software is presented in chapter 5, demonstrating the properties and features of the software implemented in chapter 4. Chapter 6 briefly touches upon a way this AD tool can be extended to and integrated with an AD machine-learning library. Illustrating potential mixing of the finite-element and machine-learning models. Finally, chapter 7 summarizes the results and provides a small discussion on future work.

1.2 Notation and assumptions

Because derivatives are a recurrent theme throughout this thesis, it is useful to clarify some notation. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be differentiable at a point $x \in \mathbb{R}^n$, then we define the Jacobian matrix as the $m \times n$ matrix with entries

$$(J_{f,x})_{i,j} = \frac{\partial f_i(x)}{\partial x_j},$$

for each $i = 1, 2, ..., m$ and $j = 1, 2, ..., n$. Where $f_i(x)$ is the i-th component of the output vector $f(x)$, and x_j is the j-th component of the input vector x. Sometimes $J_{f,x}$ will denote the linear transformation induced by the Jacobian matrix at x.

Furthermore, throughout this thesis it is assumed that the discretized functional $J(u, m) \in \mathbb{R}$ and the discretized PDE operator $F(u, m)$ are continuously differentiable. And that the discrete PDE operator $F(u, m)$ yields a unique $u \in U$ for each $m \in M$ for some suitable Banach spaces U and M.

Chapter 2

Background

In this chapter we will present necessary background for the following chapters. First, in section 2.1 we give a brief review of different applications of gradients in science and engineering. Then, in section 2.2 we introduce AD as a way to obtain these gradients, and develop the necessary mathematical tools that aims to ease the transition to code. Then, in section 2.3 we give a brief introduction to the finite element method for discretizing partial differential equations. Finally, in section 2.4 we derive the adjoint equations and then show how AD will be applied to obtain these equations.

2.1 Where are gradients useful?

Before we get into the technical details of deriving gradients, it can be useful to take a step back and ask why we even need them. This section aims to give a brief background on different ways gradients are used in science and engineering.

2.1.1 PDE constrained optimization

As mentioned in the introduction, gradients can be used for solving PDE-constrained optimization problems. We mentioned design problems, in which optimization is used to find parameters that optimize the design for certain properties. However, other kinds of problems can also be formulated as PDE-constrained optimization problems.

Inverse problems and variational data assimilation are other types of PDE-constrained optimization problems. These problems aim to find parameters which fit the simulation to observations. A classical example is that of a meteorologist who wishes to predict the weather tomorrow. Weather forecasting can be considered an initial-value problem, for which we start at an initial atmospheric state, and use physical models to simulate the state forwards in time. Of course, it is not an easy task to obtain a complete initial global atmospheric state. The observations are instead just partially complete as they are obtained from weather stations. Because of this, variational data assimilation can be used to find an initial state that best fits the observations. The objective functional J is then chosen to be a measure for the misfit between the simulation and the observations, while m is the unknown initial condition.

Since the initial condition typically consists of more than 10^6 degrees of freedom, the adjoint method becomes the only feasible approach for gradient-based optimization [41]. In section 5.2 we will explore an inverse problem identifying apparent diffusion coefficients in a diffusion equation modelling the spreading of contrast agents in patient magnetic resonance imaging (MRI) brain scans.

2.1.2 Sensitivity Analysis

Sometimes the values of the gradient can be of interest. In sensitivity analysis the aim is to analyse which parameters the functional is most sensitive to. This can be used to estimate uncertainty and identify parameters which matter the least and the most for an accurate model. The parameters that have little effect on the output can be removed prior to doing optimization, or parameters that have the most effect on the output can give indication that they must be measured more accurately. For example, the bottom topography of the Drake passage is not fully and accurately mapped. By using the Navier-Stokes equations and with J measuring the net transport through the Drake passage, the gradient can reveal where the transport is most sensitive to the topography [31].

2.1.3 Machine learning

Optimization problems in general, not just with a PDE-constraint, are problems where gradients can be useful. Machine learning algorithms often aim at minimizing a loss function, for which gradient-based optimization algorithms are effective. For example, in deep neural networks gradients are used to optimize the weights (parameters) of the network so that it minimizes the error between predicted outputs and desired outputs. Stochastic gradient descent has been proven very proficient in tuning the parameters of these deep neural networks, such as in the image classifier ImageNet [28].

2.2 Algorithmic Differentiation

Now that the importance of gradients has been established, a technique for computing them is necessary. Algorithmic differentiation is a technique for automatically computing derivatives of computer programs. This involves decomposing the program into a sequence of elementary functions, like sin, exp, addition, or multiplication, for which the symbolic derivative is known, and by automatically applying the chain rule to obtain the derivative of the function. In other words, consider a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$ that can be decomposed as

$$f(x) = g_k \circ g_{k-1} \circ \cdots \circ g_2 \circ g_1(x) \tag{2.1}$$

with $g_i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_{i+1}}$ for $i = 1, 2, ..., k$ and $n_1 = n$ and $n_{k+1} = m$. Then differentiating this with respect to x yields, by the chain rule

$$\frac{df(x)}{dx} = \frac{\partial g_k(w_k)}{\partial w_k} \frac{\partial g_{k-1}(w_{k-1})}{\partial w_{k-1}} \cdots \frac{\partial g_2(w_2)}{\partial w_2} \frac{\partial g_1(x)}{\partial x} \tag{2.2}$$

where we have the intermediate solutions

$$w_i(x) = g_{i-1} \circ g_{i-2} \circ \cdots \circ g_2 \circ g_1(x), \quad i = 2, 3, ..., k$$
$$w_1(x) = x$$

The right hand side of (2.2) is just a series of Jacobian matrix multiplications. Thus, to compute the derivative of f with respect to x, we only need to know the symbolic derivative of each function g_i with respect to its direct input. Algorithmic differentiation is effective because even for computing complex models a program essentially performs simple elementary operations in a specific order. There are two main methods of evaluating the right hand side of (2.2). Either by *tangent linear mode*, or more widely known as forward mode, or by *adjoint mode*, also known as reverse mode.

2.2.1 Tangent linear mode

In the tangent linear mode (TLM) we evaluate the right hand side of equation (2.2) in the same order as (2.1), i.e from right to left. First one chooses a seed direction $\delta x \in \mathbb{R}^n$, then multiplying equation (2.2) from the right with this seed yields the tangent linear model. Let $J_{f,x} \in L(\mathbb{R}^n, \mathbb{R}^m)$ be the linear transformation defined by the Jacobian matrix of f at x. Then, if for each $i = 1, 2, ..., k$ we assume that $g_i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_{i+1}}$ is continuously differentiable at w_i, then its Jacobian matrix induces the linear transformation $J_{g_i,w_i} \in L(\mathbb{R}^{n_i}, \mathbb{R}^{n_{i+1}})$. And so the TLM computes

$$J_{f,x}(\delta x) = J_{g_k,w_k} \circ J_{g_{k-1},w_{k-1}} \circ \cdots \circ J_{g_2,w_2} \circ J_{g_1,x}(\delta x) \tag{2.3}$$

Thus, if δx is a unit vector, and $m = 1$, then TLM equates to computing the scalar directional derivative at x:

$$J_{f,x}(\delta x) = \nabla f \cdot \delta x \tag{2.4}$$

Furthermore, if δx is a unit vector along one of the axes, we obtain the scalar partial derivative along that axis. Similarly for $m > 1$, the resulting computation is a vector $J_{f,x}(\delta x) \in \mathbb{R}^m$ with each entry being the directional derivative of the corresponding component of f along the unit vector δx.

To obtain the full Jacobian matrix in the TLM, one would need to run the computations n times with the directional seed as each Cartesian basis vector for $\mathbb{R}^n$. In other words, it would require n sweeps over the tangent linear model. In comparison, as we will see later, the adjoint mode requires m sweeps over the adjoint model to obtain the full Jacobian. This means that TLM is a preferable method when $m \gg n$, while the adjoint mode is preferable when $m \ll n$.

2.2.2 TLM Example

As an example, consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by

$$f(x_1, x_2) = \sin(x_1 x_2)$$

The Jacobian matrix of this function is

$$J_{f,(x_1,x_2)} = \begin{pmatrix} x_2 \cos(x_1 x_2) & x_1 \cos(x_1 x_2) \end{pmatrix}$$

Given a seed direction $\delta x = (\delta x_1, \delta x_2)^T \in \mathbb{R}^2$ and a point $x \in \mathbb{R}^2$, a TLM sweep computes

$$J_{f,(x_1,x_2)}(\delta x_1, \delta x_2) = \begin{pmatrix} x_2 \cos(x_1 x_2) & x_1 \cos(x_1 x_2) \end{pmatrix} \begin{pmatrix} \delta x_1 \\ \delta x_2 \end{pmatrix}$$

The function f can be decomposed into two functions

$$f(x_1, x_2) = g_2(g_1(x_1, x_2))$$

where $g_1(x_1, x_2) = x_1 x_2$ and $g_2(w_2) = \sin(w_2)$. For this example, we choose $\delta x = (0, 1)$ and wish to evaluate the derivative of f at some given point $x = (x_1, x_2)$. When the function f is evaluated, the intermediate values are computed.

$$\begin{aligned} w_2 &= x_1 x_2 \\ w_3 &= \sin(w_2) \\ f &= w_3 \end{aligned}$$

The TLM computations are as follows

$$\begin{aligned} \delta w_2 &= J_{g_1,x}(\delta x) = \begin{pmatrix} x_2 & x_1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = x_1 \\ \delta w_3 &= J_{g_2,w_2}(\delta w_2) = \cos(w_2)\delta w_2 = \cos(x_1 x_2) x_1 \\ J_{f,x}(\delta x) &= \delta w_3 = x_1 \cos(x_1 x_2) \end{aligned}$$

Note that the values computed are all numerical. Thus, one sweep only computes the directional derivative at a single point. For this example, if $x = (1, 0)^T$ then the TLM computations would result in $\delta w_3 = 1$.

2.2.3 Adjoint mode

With the adjoint mode one instead chooses a seed δy in the codomain of f, $\delta y \in \mathbb{R}^m$. Multiplying (2.2) from the left with δy^T, i.e the transpose of the seed, and computing from left to right, equates to the computations done in the adjoint mode. Specifically, if $J_{f,x}$ is the Jacobian matrix of f at point x, then this would mean the computation of

$$\delta y^T J_{f,x} \in \mathbb{R}^{1 \times n} \tag{2.5}$$

Choosing δy as a Cartesian basis vector for $\mathbb{R}^m$ would result in the corresponding row in the Jacobian matrix $J_{f,x}$. This highlights the main difference between the TLM and the adjoint mode: the adjoint mode extracts rows while the TLM extracts columns of the Jacobian matrix $J_{f,x}$. Of course in reality, it is a weighted sum of the rows or columns.

We will however be a little bit more precise. The product in (2.5) is a row vector, but we are actually interested in producing column vectors because we view the input x as column vectors. Thus the operator we are actually employing is the transpose (or adjoint) of the Jacobian matrix.

Definition 2.2.1 *The adjoint of the linear operator $J_{f,x} : \mathbb{R}^n \to \mathbb{R}^m$ is defined as $J^*_{f,x} : \mathbb{R}^m \to \mathbb{R}^n$ such that for each $\delta x \in \mathbb{R}^n$ and each $\delta y \in \mathbb{R}^m$*

$$\langle J_{f,x}(\delta x), \delta y \rangle_{\mathbb{R}^m} = \left\langle \delta x, J^*_{f,x}(\delta y) \right\rangle_{\mathbb{R}^n}$$

where $\langle \cdot, \cdot \rangle_{\mathbb{R}^m}$ and $\langle \cdot, \cdot \rangle_{\mathbb{R}^n}$ are the dot products in $\mathbb{R}^m$ and $\mathbb{R}^n$ respectively.

It can be shown that the transpose of the Jacobian matrix $J^T_{f,x}$ induces the adjoint operator $J^*_{f,x}$, see for example [38, Sect 2.2]. It follows that the adjoint operator can be decomposed to

$$J^*_{f,x}(\delta y) = J^*_{g_1,x} \circ J^*_{g_2,w_2} \circ \cdots \circ J^*_{g_{k-1},w_{k-1}} \circ J^*_{g_k,w_k}(\delta y)$$

If $m = 1$, choosing $\delta y = 1$ results in $J^*_{f,x}(\delta y) = \nabla f$. Thus, no matter how big n is, we only need to do one sweep of the adjoint mode to get the full gradient. In general the result of the adjoint computations is the gradient of a weighted sum of the components of f

$$J^*_{f,x}(\delta y) = \nabla(\delta y^T f(x)) = \sum_{i=1}^{m} \delta y_i \nabla f_i(x)$$

2.2.4 Adjoint example

Again we consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by

$$f(x_1, x_2) = \sin(x_1 x_2)$$

Recall that the Jacobian matrix of this function is

$$J_{f,(x_1,x_2)} = \begin{pmatrix} x_2 \cos(x_1 x_2) & x_1 \cos(x_1 x_2) \end{pmatrix}$$

and consequently, given a seed $\delta y \in \mathbb{R}$ and a point $x \in \mathbb{R}^2$ one adjoint sweep computes

$$J^*_{f,(x_1,x_2)}(\delta y) = \begin{pmatrix} x_2 \cos(x_1 x_2) \\ x_1 \cos(x_1 x_2) \end{pmatrix} \delta y$$

The intermediate values are computed as before. We choose $\delta y = 1$ for this example, then the adjoint computations are

$$\begin{aligned} \delta g_2 &= J^*_{g_2,w_2}(\delta y) = \cos(w_2)\delta y = \cos(x_1 x_2) \\ \delta g_1 &= J^*_{g_1,w_1}(\delta g_2) = \begin{pmatrix} x_2 \\ x_1 \end{pmatrix} \delta g_2 = \begin{pmatrix} x_2 \cos(x_1 x_2) \\ x_1 \cos(x_1 x_2) \end{pmatrix} \\ J^*_{f,x}(\delta y) &= \delta g_1 = \nabla f \end{aligned}$$

Notice that the computation starts with the last operation and propagates backwards in comparison to the forward computation of f.

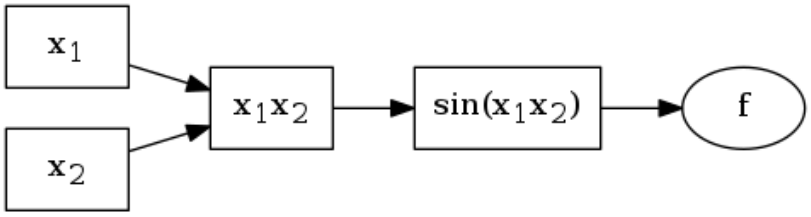


Figure 2.1: A computational graph of the function from the examples. TLM AD compute in the same direction as the graph, while the adjoint mode traverses the operations backwards starting from the output f.

2.2.5 The program structure

In TLM there was no need for remembering or storing any program structure because everything could be computed together with the computation of f. However, with adjoint mode there is a need for a structured way of tracking operations performed and storing intermediate variables. This leads to the introduction of a computational graph. This can be interpreted as a directed acyclic graph (DAG) of the operations performed in the forward computations. Figure 2.1 shows a DAG for the function f from the examples above. The adjoint mode can be thought of as initializing one of the leaf nodes (i.e. nodes which have no outgoing edges) and propagating computed values backwards in the graph.

The data structure comprising of the operations and intermediate variables is called a *tape*. Annotating the tape is the process of tracking and storing the computed operations in the tape.

As will be explained in the next section, the implementation of AD can be done in several ways. However, common for all techniques is that each operation which can make up an AD-compatible composite function of f must be augmented to compute derivatives. This augmentation can be regarded as follows. Let $g : \mathbb{R}^n \to \mathbb{R}^m$ be a function that we wish to augment with a derivative function. The tangent linear model of g is a function $g^{(1)} : \mathbb{R}^{2n} \to \mathbb{R}^m$ defined as

$$g^{(1)}(w, \delta w) = J_{g,w}(\delta w)$$

The value $\delta w \in \mathbb{R}^n$ is the tangent linear input to the function g. If g is the i-th function in (2.1), i.e. g_i for $i = 2, 3, ..., k$, then $\delta w \in \mathbb{R}^n$ is the result of

$$\delta w = J_{g_{i-1},w_{i-1}}(\delta x)$$

for some $\delta x \in \mathbb{R}^n$.

Similarly, we define the adjoint model of g as the function $g_{(1)} : \mathbb{R}^{n+m} \to \mathbb{R}^m$

$$g_{(1)}(w, \delta g) = J^*_{g,w}(\delta g) \tag{2.6}$$

The value $\delta g \in \mathbb{R}^m$ is the adjoint input to the function g. Again, if g is the i-th function in (2.1) for $i = 1, 2, ..., k-1$, then $\delta g \in \mathbb{R}^m$ is computed as

$$\delta g = J^*_{g_{i+1},w_{i+1}}(\delta y)$$

for some $\delta y \in \mathbb{R}^m$.

Note that the dimensions of the vectors δx and δy above stem from f in (2.1) and not g. The implementation of the tangent linear and adjoint model of g is entirely independent of f.

2.2.6 Main implementation approaches

There are two ways algorithmic differentiation is implemented. Either by *source code transformation* or by *operator overloading*. As the name suggests, the source code transformation approach involves transforming the program's source code, adding algorithmic differentiation routines for every addition, multiplication, as well as every call to functions like sin and exp. This approach can lend greater runtime efficiency, as the compiler is able to do code optimizations. However, it can be a challenge to implement a tool that actually performs these source code transformations automatically.

Operator overloading instead works by overloading operators like addition (+) and multiplication (*), adding a routine for computing the tangent linear or adjoint model to the overloaded version. This is also done in general for other basic functions like cos and log. Typically, one implements a new data type overloading operators on it so that they also produce the same data type. This means that if `x` is an AD data type, `y = x*x` is also an AD data type. This way, you can just send an AD floating point number into a function that does arithmetic operations on floats and, without any changes to the function itself, get back an AD data type providing the derivative. Thus this method can lead to some very simple implementations that work for a wide range of functions. However, not every programming language actually supports operator overloading. For instance, Java does not offer this feature. Furthermore, the program can not make use of compiler optimizations, and will generally be slower than a source code transformation equivalent.

2.2.7 Higher order derivatives

Algorithmic differentiation can be generalized to higher order derivatives by performing AD on the tangent linear or adjoint model. A detailed description of this process is provided in [38, Chap 3].

2.3 Finite Element Method

Because gradients are often useful in problems involving PDEs, such as PDE-constrained optimization, we now look at how PDEs can be solved numerically. Specifically, we will be introducing the Galerkin finite element method.

The finite element method (FEM) is a way of solving PDEs by approximating functions over a domain. Let V be the target function space of functions over the domain for which we wish to use the finite element method. The general idea is to define a finite dimensional vector space, called a finite element space, $V_h = \text{span}\{\phi_1(x), \phi_2(x), ..., \phi_k(x)\}$ such that any $u_h \in V_h$ can be

represented as a linear combination

$$u_h(x) = \sum_{i=1}^{k} \mathcal{U}_i \phi_i(x)$$

After defining the space, it would just remain to find the best approximation $u_h \in V_h$ of $u \in V$. In finite element spaces the domain is divided into a collection of subdomains, or cells, with each basis function ϕ_i being defined locally as a polynomial on one or more of these cells. In other words, the basis functions $\{\phi_i\}_{i=1}^k$ have compact support. On each cell we place nodes, and some nodes will be vertices between multiple cells. In total there should be k nodes, each corresponding to one basis function ϕ_i. This means that for node numbered i located at the coordinates x_i, $\phi_i(x_i) = 1$ and $\phi_j(x_i) = 0$ for $j \neq i$.

Let $\Omega \subset \mathbb{R}^n$ be open and bounded, with boundary $\partial\Omega$. As an example, we consider the poisson equation

$$\begin{aligned} -\Delta u &= f, \quad \text{in } \Omega \\ \nabla u \cdot \nu &= g_N, \quad \text{on } \partial\Omega_N \\ u &= g_D, \quad \text{on } \partial\Omega_D \end{aligned}$$

where $f : \Omega \to \mathbb{R}$, $g_N : \partial\Omega_N \to \mathbb{R}$ and $g_D : \partial\Omega_D \to \mathbb{R}$ are given functions and $u : \Omega \to \mathbb{R}$ is the unknown solution. With $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$ and $\partial\Omega_D \cap \partial\Omega_N = \emptyset$. $\partial\Omega_D$ is the Dirichlet boundary and $\partial\Omega_N$ is the Neumann boundary. ν is the outward pointing normal vector on $\partial\Omega$.

As it is not always easy or even possible to find a $u \in C^2(\Omega)$ solution to this PDE, we may wish to find a solution in a wider space V. To this end we choose a suitable test space V' and trial space V, and multiply the equation with a test function $v \in V'$, and integrate over the domain. Then we apply integration by parts

$$\begin{aligned} -\int_\Omega \Delta u v \, dx &= \int_\Omega f v \, dx \\ \implies \int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \nabla u \cdot \nu v \, dS &= \int_\Omega f v \, dx \end{aligned}$$

Thus, the problem becomes: find a $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \nabla u \cdot \nu v \, dS = \int_\Omega f v \, dx, \quad \forall v \in V'$$

This is called the weak or variational formulation of the problem. A natural choice for V and V' is the Sobolev space

$$H^1 = \{v : \Omega \to \mathbb{R} \,|\, v \in L^2(\Omega), \frac{\partial v}{\partial x_i} \in L^2(\Omega) \; \forall i = 1, 2, .., n\}$$

where $\frac{\partial v}{\partial x_i}$ for each $i = 1, 2, ..., n$ is the weak partial derivatives of v. For more on Sobolev spaces and weak derivatives, see the book by Evans [9, Chap 5]. Furthermore, to strongly enforce the Dirichlet boundary conditions we choose the function spaces

$$\begin{aligned} V = H_g^1 &= \{u \in H^1(\Omega) \,|\, u = g \text{ on } \partial\Omega_D\} \\ V' = H_0^1 &= \{v \in H^1(\Omega) \,|\, v = 0 \text{ on } \partial\Omega_D\} \end{aligned}$$

Then our problem becomes: find $u \in H_g^1$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega_N} g_N v \, dS = \int_\Omega f v \, dx, \quad \forall v \in H_0^1 \tag{2.7}$$

However, solving (2.7) numerically when the trial space and test space are infinite dimensional is infeasible. Instead we introduce a finite element approximation. Let $V_{0,h} \subset H_0^1$ be a finite element space of dimension $k \in \mathbb{N}$, spanned as before by the basis functions $\{\phi_1(x), \phi_2(x), ..., \phi_k(x)\}$. For the trial space, we additionally must enforce the Dirichlet boundary condition g_D. We define an additional set of basis functions $\{\phi_{k+1}(x), \phi_{k+2}(x), ..., \phi_{k+k_d}(x)\}$ for which we interpolate the Dirichlet boundary condition g_D. Then the discretized trial space $V_{g,h}$ is spanned by the basis functions $\{\phi_1(x), ..., \phi_k(x), \phi_{k+1}(x), ..., \phi_{k+k_d}(x)\}$ and any solution in $u_h \in V_{g,h}$ can be represented as

$$u_h(x) = \sum_{i=1}^{k} \mathcal{U}_i \phi_i(x) + \sum_{i=k+1}^{k+k_d} g_{h,i} \phi_i(x) \tag{2.8}$$

Because the coefficient $g_{h,i}$ is determined by just interpolating g_D at the node of ϕ_i for $i = k+1, k+2, ..., k+k_d$, we only need to determine the coefficient u_j for $j = 1, 2, ..., k$. We say that the problem has k degrees of freedom. By reformulating (2.7) with the spatially discretized function spaces, we obtain the discrete variational formulation: find $u_h \in V_{g,h}$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \int_{\partial\Omega_N} g_N v_h \, dS = \int_\Omega f v_h \, dx, \quad \forall v_h \in V_{0,h} \tag{2.9}$$

Or more explicitly equation (2.9) is equivalent to

$$\sum_{j=1}^{k+k_d} \mathcal{U}_j \int_\Omega \nabla \phi_j \cdot \nabla \phi_i \, dx = \int_\Omega f \phi_i \, dx + \int_{\partial\Omega_N} g_N \phi_i \, dS, \quad \forall i = 1, 2, ..., k \tag{2.10}$$

The Dirichlet boundary condition is enforced with the k_d additional equations

$$\mathcal{U}_j = g_{h,j}, \quad \forall j = k+1, k+2, ..., k+k_d \tag{2.11}$$

From (2.10) and (2.11) we obtain the linear system

$$A\mathcal{U} = b$$

where $\mathcal{U} = (\mathcal{U}_1, \mathcal{U}_2, ..., \mathcal{U}_k, \mathcal{U}_{k+1}, ..., \mathcal{U}_{k+k_d})^T$, and $b \in \mathbb{R}^{k+k_d}$ with

$$\begin{aligned} b_i &= \int_\Omega f \phi_i \, dx + \int_{\partial\Omega_N} g_N \phi_i \, dS, \quad \forall i = 1, 2, ..., k \\ b_i &= g_{h,i}, \quad \forall i = k+1, k+2, ..., k+k_d \end{aligned}$$

The matrix A is defined in block matrix form as

$$A = \begin{pmatrix} A^{00} & A^g \\ 0 & I \end{pmatrix}$$

with $A^{00}_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, dx$ for $i, j = 1, 2, ..., k$. $A^g_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, dx$ for $i = 1, 2, ..., k$ and $j = 1, 2, k_d$. While I is the $k_d \times k_d$ identity matrix.

If it was not for the Dirichlet boundary condition equations (2.11), the matrix A would have been symmetric, i.e A^{00} is symmetric. To maintain symmetry of the linear system, one could alternatively reduce the system to only A^{00} by only considering $\mathcal{U} = (\mathcal{U}_1, \mathcal{U}_2, ..., \mathcal{U}_k)^T$ and consequently reducing the system to

$$A^{00}\mathcal{U} = b - A^g g$$

where b is reduced to only the first k entries and $g = (g_{h,k+1}, g_{h,k+2}, ..., g_{h,k+k_d})$.

2.4 The adjoint equations

In order to obtain the gradient of a reduced functional in PDE-constrained optimization, the derivative of the PDE solution with respect to the parameters must be computed. This leads to the derivation of the adjoint equation, which are presented in this section.

Consider a PDE on the form

$$F(u, m) = 0 \tag{2.12}$$

where $m \in M$ is a parameter of F and $u \in U$ is the solution, for some Banach spaces M and U. We assume F is continuously Fréchet differentiable, and has a unique solution $u(m) \in U$ for each $m \in M$. Furthermore we assume that the linearisation $\partial F(u(m), m)/\partial u$ of the PDE operator is invertible. Then by the implicit function theorem $u : M \to U$, $m \mapsto u(m)$ is continuously Fréchet differentiable [19, Sect 1.4.2].

Additionally, we are interested in the derivative of a functional

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

with respect to m, when (2.12) is satisfied. Again assuming that J is continuously Fréchet differentiable. Using the chain rule, observe that

$$\frac{dJ}{dm} = \frac{\partial J}{\partial u}\frac{du}{dm} + \frac{\partial J}{\partial m} \tag{2.13}$$

As J is usually explicitly defined through an analytical formula, deriving $\partial J/\partial u$ and $\partial J/\partial m$ is typically not a problem. But how to compute du/dm is not immediately obvious.

However, u is implicitly defined through m by the relationship $F(u(m), m) = 0$. We thus use implicit differentiation with respect to m to obtain

$$\begin{aligned} &\frac{\partial F(u(m), m)}{\partial u}\frac{du}{dm} + \frac{\partial F(u(m), m)}{\partial m} = 0 \\ \implies &\frac{\partial F(u(m), m)}{\partial u}\frac{du}{dm} = -\frac{\partial F(u(m), m)}{\partial m} \end{aligned} \tag{2.14}$$

This yields a linear equation for du/dm, and is called the *tangent linear equation*. Because $F(u, m)$ is always going to be explicitly given, we should always be able to obtain the terms $\partial F(u(m), m)/\partial u$ and $\partial F(u(m), m)/\partial m$.

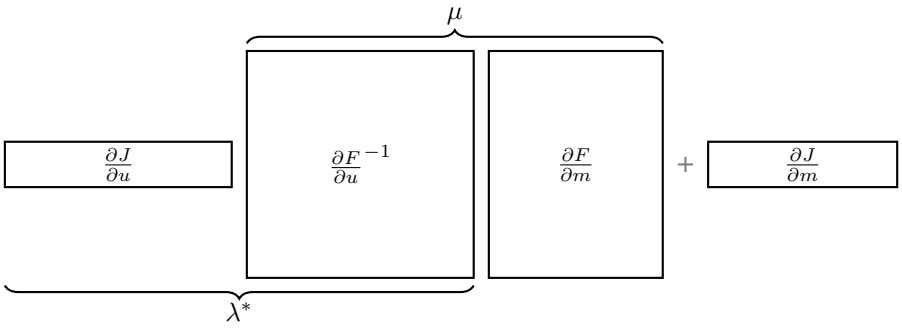


Figure 2.2: The finite dimensional composition of (2.15). With the tangent linear equation, finding μ involves solving n linear systems. The adjoint equation only requires solving one linear system as long as the codomain of J is one-dimensional.

$\partial F(u(m), m)/\partial u$ is assumed invertible, thus we write

$$\frac{du}{dm} = -\frac{\partial F(u(m), m)}{\partial u}^{-1} \frac{\partial F(u(m), m)}{\partial m}$$

Inserting this into (2.13) leads to

$$\frac{dJ}{dm} = -\frac{\partial J}{\partial u} \frac{\partial F(u(m), m)}{\partial u}^{-1} \frac{\partial F(u(m), m)}{\partial m} + \frac{\partial J}{\partial m} \tag{2.15}$$

Now we consider the hermitian adjoint of this linear operator

$$\frac{dJ}{dm}^* = -\frac{\partial F(u(m), m)}{\partial m}^* \frac{\partial F(u(m), m)}{\partial u}^{-*} \frac{\partial J}{\partial u}^* + \frac{\partial J}{\partial m}^*$$

And define

$$\begin{aligned} \lambda &= \frac{\partial F(u(m), m)}{\partial u}^{-*} \frac{\partial J}{\partial u}^* \\ \implies \frac{\partial F(u(m), m)}{\partial u}^* \lambda &= \frac{\partial J}{\partial u}^* \end{aligned} \tag{2.16}$$

Equation (2.16) is called the *adjoint equation*. The solution λ is called the adjoint solution. For notational convenience we will from now on denote the solution of the tangent linear equation as $\mu = du/dm$.

For a more thorough discussion on adjoints, see [19, Sect 1.6].

2.4.1 Comparing the tangent linear and adjoint equation

To compare the two methods we will be considering a finite dimensional case. Let $U = \mathbb{R}^k$ and $M = \mathbb{R}^n$ for some $k, n \in \mathbb{N}$. Then we have $u : \mathbb{R}^n \to \mathbb{R}^k$, $m \mapsto u(m)$. In figure 2.2 below we illustrate how the right hand side of (2.15) is composed.

Observe that to find $\mu \in \mathbb{R}^{k \times n}$ requires solving n linear systems. While $\lambda \in \mathbb{R}^{k \times 1}$ only requires solving one linear system, where 1 is caused by the assumption that $J(u(m), m) \in \mathbb{R}$. If instead $J(u(m), m) \in \mathbb{R}^l$, then λ would be a $k \times l$ matrix, requiring solving l linear systems. This is comparable to the tangent linear mode and adjoint mode AD described in section 2.2.

2.4.2 Applying AD to PDE

We now try to fit the above equations into an AD perspective. Consider a computer program that involves solving a PDE $F(u, m)$ with the same assumptions as above. Further, assume that both m and $u(m)$ can be represented as finite dimensional real vectors. More specifically, assume that $m \in \mathbb{R}^n$ and $u(m) \in \mathbb{R}^k$. Given that this PDE has to be solved numerically, this is a reasonable assumption. We can now consider a general function $f : \mathbb{R}^q \to \mathbb{R}^p$, decomposed as

$$f(x) = g \circ u \circ m(x)$$

where we let $m : \mathbb{R}^q \to \mathbb{R}^n$ and $g : \mathbb{R}^k \to \mathbb{R}^p$ be some functions for which we can apply normal AD. As before, using the chain rule, the derivative is expressed as

$$\frac{df(x)}{dx} = \frac{\partial g(w_3)}{\partial w_3} \frac{\partial u(w_2)}{\partial w_2} \frac{\partial m(x)}{\partial x}$$

with intermediate solutions $w_3 = u \circ m(x)$, $w_2 = m(x)$. As described in section 2.2, applying tangent linear mode AD to f would involve choosing a seed direction $\delta x \in \mathbb{R}^q$ and computing

$$J_{f,x}(\delta x) = J_{g,w_3} \circ J_{u,w_2} \circ J_{m,x}(\delta x)$$

As both $J_{m,x}$ and J_{g,w_3} can be implemented directly through AD techniques described before, we take a closer look only at J_{u,w_2}.

By definition, J_{u,w_2} should be a linear transformation defined by the Jacobian matrix of u. That is just du/dm, i.e. the solution of the tangent linear equation. However, notice that we do not actually need the whole du/dm matrix for each tangent linear AD run. Instead

$$\begin{aligned} \frac{\partial u(w_2)}{\partial w_2} &= -\frac{\partial F(u(w_2), w_2)}{\partial u}^{-1} \frac{\partial F(u(w_2), w_2)}{\partial w_2} \\ \implies J_{f,x}(\delta x) &= J_{g,w_3} \circ \frac{\partial F(u(w_2), w_2)}{\partial u}^{-1} \left(-\frac{\partial F(u(w_2), w_2)}{\partial w_2} J_{m,x}(\delta x) \right) \end{aligned}$$

Hence, with m denoting $m(x)$ for ease of notation, the AD tangent linear equation can be written as

$$\frac{\partial F(u(m), m)}{\partial u} \mu_{\delta m} = -\frac{\partial F(u(m), m)}{\partial m} \delta m \tag{2.17}$$

where

$$\begin{aligned} \delta m &= J_{m,x}(\delta x) \\ \mu_{\delta m} &= \frac{du}{dm} \delta m = J_{u,w_2} \circ J_{m,x}(\delta x) \end{aligned}$$

Here $\delta m \in \mathbb{R}^n$, thus the right hand side of (2.17) is a vector in $\mathbb{R}^k$. This means that we now only solve one linear system for one tangent linear mode AD run. However, we still only compute the directional derivative of du/dm, hence we

need to run the tangent linear mode AD n times (and by that solve n linear systems) to assemble the whole Jacobian $J_{f,x} = df/dx$.

Similarly, for the adjoint mode AD we choose a seed $\delta y \in \mathbb{R}^p$ and compute

$$J^*_{f,x}(\delta y) = J^*_{m,x} \circ J^*_{u,w_2} \circ J^*_{g,w_3}(\delta y)$$

Again we only need to treat J^*_{u,w_2} in a special manner. As already established, $J_{u,w_2} = du/dm$, thus $J^*_{u,w_2} = (du/dm)^*$.

$$\begin{aligned}
&\frac{du}{dm}^* = -\frac{\partial F(u(w_2), w_2)}{\partial w_2}^* \frac{\partial F(u(w_2), w_2)}{\partial u}^{-*} \\
\implies &J^*_{f,x}(\delta y) = J^*_{m,x} \circ \left(-\frac{\partial F(u(w_2), w_2)}{\partial w_2}^* \right) \frac{\partial F(u(w_2), w_2)}{\partial u}^{-*} J^*_{g,w_3}(\delta y)
\end{aligned}$$

Hence we end up with the AD adjoint equation

$$\frac{\partial F(u(m), m)}{\partial u}^* \lambda_{\delta u} = \delta u^* \tag{2.18}$$

where

$$\delta u^* = J^*_{g,w_3}(\delta y)$$

To obtain $\lambda_{\delta u}$ we only need to solve one linear system as $\delta u^* \in \mathbb{R}^k$. However, to actually compute the action of J_{u,w_2} on δu^* we additionally need to do a matrix-vector multiplication. Hence we end up with

$$J^*_{u,w_2}(\delta u) = -\frac{\partial F(u(w_2), w_2)}{\partial w_2}^* \lambda_{\delta u} \tag{2.19}$$

Finally, note that any explicitly given function $y = f(x)$, with $f : \mathbb{R}^n \to \mathbb{R}^m$, can be represented on the form $F(y, x) = y - f(x) = 0$. Then the terms are $\partial_y F = \partial_y F^* = I$, $\partial_x F = J_{f,x}$ and $\partial_x F^* = J^*_{f,x}$. Hence, the theory derived here is entirely analogous with tangent linear/forwards and adjoint/reverse mode AD.

2.4.3 Applying Dirichlet boundary conditions

Often the PDE $F(u, m)$ is prescribed some Dirichlet boundary conditions g. We will now discuss how to apply these in the resulting linear systems for the tangent linear and adjoint equations. Assume that $F(u, m)$ has been discretized and assembled into a possibly non-linear system

$$A(u, m)u = b(u, m)$$

where u is the coefficients u_i of the discrete solution of the PDE, $A(u, m)$ is the system matrix and $b(u, m)$ the source term vector. Also assume that Dirichlet boundary conditions have been prescribed to this system through the equations $u_i = g_i$ for suitable i as described in 2.3. Defining $F(u, m) = A(u, m)u - b(u, m)$, the resulting tangent linear equation (2.14) is

$$(A(u, m) + C(m, u) - B(m, u))\mu = -\frac{\partial F}{\partial m} \tag{2.20}$$

where $C = \frac{\partial A}{\partial u}u$ and $B = \frac{\partial b}{\partial u}$. Given that g does not depend on m, we have that the boundary equations reduce to $\mu_i = 0$. Let $G = A + C - B$ be the tangent linear equation matrix, and $D = \partial F/\partial m$ the matrix on the right hand side of (2.20). The matrix can be considered on block form

$$G = \begin{pmatrix} G^{00} & G^{0g} \\ 0 & I \end{pmatrix}$$

resulting in equation 2.20 being on the form

$$\begin{pmatrix} G^{00} & G^{0g} \\ 0 & I \end{pmatrix} \begin{pmatrix} \mu^0 \\ \mu^g \end{pmatrix} = - \begin{pmatrix} D^0 & D^{01} \\ 0 & 0 \end{pmatrix}$$

Moving on to the adjoint equation (2.16), the system matrix is

$$G^* = \begin{pmatrix} (G^{00})^* & 0 \\ (G^{0g})^* & I \end{pmatrix}$$

and the adjoint equation takes the form

$$\begin{pmatrix} (G^{00})^* & 0 \\ (G^{0g})^* & I \end{pmatrix} \begin{pmatrix} \lambda^0 \\ \lambda^g \end{pmatrix} = \begin{pmatrix} (\delta u^0)^* \\ (\delta u^g)^* \end{pmatrix} \tag{2.21}$$

The values of λ^g do not matter for the interior adjoint solution λ^0. Thus, by modifying the system as if it was prescribed homogeneous boundary conditions yield

$$\begin{pmatrix} (G^{00})^* & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \lambda^0 \\ \lambda^g \end{pmatrix} = \begin{pmatrix} (\delta u^0)^* \\ 0 \end{pmatrix}$$

Often λ^0 is enough because, as seen above, $\frac{\partial F}{\partial m}^* = D^* = \begin{pmatrix} (D^0)^* & 0 \end{pmatrix}$ and so

$$\frac{\partial F}{\partial m}^* \lambda = \begin{pmatrix} (D^0)^* & 0 \end{pmatrix} \begin{pmatrix} \lambda^0 \\ \lambda^g \end{pmatrix} = (D^0)^* \lambda^0$$

If $m = g$, i.e m is the prescribed Dirichlet boundary condition, then the boundary equations reduce to $\mu_i = 1$ and so the lower block matrix of D is not all zero, requiring λ^g. When λ^0 has been obtained, one can find λ^g by computing

$$\lambda^g = (\delta u^g)^* - (G^{0g})^* \lambda^0$$

Chapter 3

A generic AD framework

In this chapter we implement a generic AD framework that works by overloading. These implementations form the python package pyadjoint. This software is open source and available at `https://bitbucket.org/dolfin-adjoint/pyadjoint`, and still in active development.

Because pyadjoint is a generic framework, we differentiate between two interfaces: implementation interface and user interface. Implementation is the step of applying the framework to another python package. This requires a developer with knowledge of both the workings of pyadjoint and the software for which pyadjoint should be applied. In section 3.1 we go through the implementation details of the framework. Once pyadjoint has been applied to a software package, the end user has access to the pyadjoint user interface together with the target software package. This is the user interface, and is described in section 3.2. Then, in section 3.3 limitations of the current pyadjoint implementation is discussed. Lastly, section 3.4 summarises the work presented in this chapter.

3.1 Implementation

The choice programming language falls naturally on Python. It is the main (or most used) programming interface for the FEM framework FEniCS [29], which is our primary target for AD application.

Our AD implementation builds on-top of the software package it is applied to. This means that it enhances a package by augmenting functions and types with extra functionality such as AD. We call the augmented function/type an overloaded function/type. Because the framework is generic, it provides very little value without some work. Specifically, it is intended to be applied to Python modules that can provide some higher level of abstraction for the most intensive computations. By itself, pyadjoint only overloads the type `float`. A more detailed discussion on the limitations of pyadjoint is presented in section 3.3.

3.1.1 The core annotation classes

Here we present the core classes for keeping track of the forward computations. If we consider the computational graph illustration in figure 2.1 then it

becomes apparent that we need to represent three key concepts in our code: inputs/outputs, functions, and the tape. We start off by going through the implementation of these core concepts.

First of all we need an object that can store the tape. For this purpose, we define the class `Tape`. It holds onto a simple python list, and also provides methods for performing operations on the list. One example of such a method is `evaluate_adj`, which initiates the reverse mode AD.

The functions/operations are represented on the tape as `Block` objects. These objects are instances of `Block` subclasses. Since the abstract[1] class `Block` only provides the interface needed for other components of pyadjoint to work, the subclasses contain the actual implementation details needed to differentiate the function.

```
1  class Block(object):
2      """Base class for all Tape Block types.
3
4      Each instance of a Block type represents an elementary
                                        operation in the
5      forward model.
6
7      Abstract methods
8          evaluate_adj
9          recompute
10
11     """
12     def evaluate_adj(self):
13         """This method must be overridden.
14
15         The method should implement a routine for evaluating the
                                            adjoint model of the
                                            block.
16
17         """
18         raise NotImplementedError
19
20     def recompute(self):
21         """This method must be overridden.
22
23         The method should implement a routine for recomputing the
                                            block in the forward
                                            computations.
24
25         """
26         raise NotImplementedError
```

Hence, each function needs its own block subclass, and that class must implement methods like `evaluate_adj` and `recompute` seen in the abstract `Block` class above. Before we delve into how such a subclass can be implemented, we first introduce the remaining core types in pyadjoint.

We now consider something to represent the input and output of the blocks. To perform adjoint mode AD, we are dependent on being able to determine the inputs and outputs of every relevant function call. At first glance this may seem trivial. If a function takes in a `float` and produces some output, say `3*x`, then the output is a new `float` and we can store references to these

[1] The `Block` class is not precisely abstract. Indeed you can skip implementing certain methods, and pyadjoint will only crash when you use a functionality that depends on that method.

objects as the values of the block's input and output. However, this only works when the objects being handled are immutable[2], and so they only have one *version*. Imagine a case where `x` is a mutable object, and it is used as input in a function `f` that mutates `x` instead of producing a new object. We say that the version changes when the values of a mutable object change. This could for example happen if we use a Python list as input `x = [1.0, 2.0]`, then `x[0] += 2.0` changes the object `x`. Here storing the memory address of `x` does not distinguish between `x` prior to the computation of `f` and the version of `x` afterwards.

The logical answer is to create a copy before the input is handled by the function, and so we define the class `BlockVariable`. Each instance of a `BlockVariable` contains at most one copy of the corresponding object, and acts as a unique identifier for a specific version of an object. Conveniently, this introduction gives us a natural place to store temporary AD values in addition to the intermediate solutions from the forward computations. In figure 3.1 the connection between blocks and block variables is illustrated.

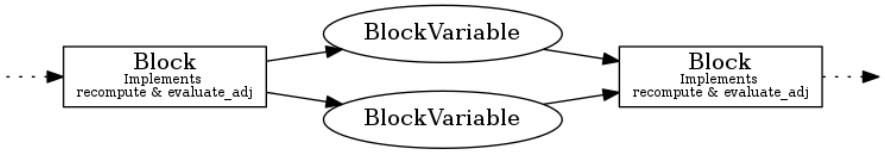


Figure 3.1: An illustration of how the types are connected on the tape. Block variables flow into the block as input, and are also produced by blocks as output. The arrows point in the direction data is flowing in the forward computations. The block variables do not know which blocks they relate to.

Each block keeps track of their inputs, called *dependencies*, and outputs. This is implemented as two lists of block variables stored in the block objects. Now we need some way of connecting a version of an object with its corresponding block variable object. After all, we are not going to feed the forward computations with block variables, but only with the original objects themselves. Thus we make all valid AD data types carry a reference to their current block variable, changing it whenever they become a new version.

To attach this reference, we introduce the `OverloadedType` class. Similarly to the `Block` this is also an abstract class, for which we subclass any valid AD data type. Using the multiple inheritance in Python, a subclass of `OverloadedType` should also be defined as a subclass of the original data type. That is, if overloading a class named `Vector`, we typically define a new overloaded class as

```
1 backend_Vector = Vector
2 class Vector(OverloadedType, backend_Vector):
3     def __init__(self, *args, **kwargs):
4         # Call the OverloadedType constructor
5         OverloadedType.__init__(self, *args, **kwargs)
6         # Call the original type constructor
7         backend_Vector.__init__(self, *args, **kwargs)
8     ...
```

[2]Immutable means you can't change its value after instantiation.

Class	Description
Tape	Stores the Block instances and represents the computational graph.
Block	Stores BlockVariable instances as inputs/outputs. And provides methods for recomputing and differentiating.
BlockVariable	Represents a version of an object, uniquely identifies an intermediate solution or input/ouput variable, and stores a copy of it.
OverloadedType	Represents an AD data type. Stores a reference to the current BlockVariable that identifies it. Implements functions that are required for certain AD operations.

Table 3.1: An overview of the core classes presented here.

Notice that we store a reference to the original type through `backend_Vector`. This enables us to call methods of the original type, such as the constructor.

The `OverloadedType` constructor initializes a `BlockVariable` object and stores a reference to it. A subclass of `OverloadedType` is only meant to enhance the original class with features that are useful for the AD library, and thus should still provide all features of the original class. A method for creating a copy of the object (input/ouput) is an example of one abstract method that the `OverloadedType` class defines. Basically all type specific methods that are needed for parts of the AD framework are defined as abstract methods in `OverloadedType`. In general, very few methods need to be overloaded for a specific type to work with AD. Most of the extra abstract methods can be categorized as API specific and only needed if you wish to use extra included pyadjoint functions beyond simple AD. Examples of such functions will be shown later.

3.1.2 Annotating the forward computations

Now that the core types have been introduced, we consider how the Blocks should be populated with information and how they are put on the tape. This is achieved by creating new functions with the same name and added pyadjoint specific code. For instance, consider the case where we wish to provide annotation for the function `target_function`, then the overloaded version would look like this:

```
1  backend_target_function = target_function
2  def target_function(*args, **kwargs):
3      annotate = annotate_tape(kwargs)
4      if annotate:
5          tape = get_working_tape()
6          block = TargetFunctionBlock(*args, **kwargs)
7          tape.add_block(block)
8
9      with stop_annotating():
10         output = backend_target_function(*args, **kwargs)
```

```
11      output = create_overloaded_object(output)
12
13      if annotate:
14          block.add_output(output.create_block_variable())
15
16      return output
```

On the first line we store a reference to the original (backend) function. Then we define a new function with the same name which will overwrite the original function in the current namespace. Inside the overloaded version we first check if the tape should be annotated. If the dictionary `kwargs` has the key "annotate" set to `False`, or if annotation is somehow globally disabled at the moment, the overloaded function should behave almost exactly as the original. If however, `annotate_tape(kwargs)` return `True`, then we initialise a corresponding Block instance and add it to the tape in lines 5-7. We then use a context manager to temporarily disable annotation while we execute the original function. Assuming `target_function` returns a single output variable, we must transform this into a corresponding `OverloadedType` instance to ensure that the data types exposed to the user have AD support, even if we were not currently annotating the call itself. This is done through the call to `create_overloaded_object` on line 11. Apart from this initialisation, the annotation-disabled function behaves exactly the same as the original. If annotation is enabled, we also have to assign a new version (`BlockVariable`) to the output object, and store this in the block instance as an output.

This simple example covers almost every function which returns one variable. If the function returned multiple variables, the `create_overloaded_object` calls and `block.add_output` would have to be called on every single output.

3.1.3 Example block implementation

As an example consider a function `euclidean_norm` that takes in an n-dimensional vector and computes the Euclidean norm:

```
1  def euclidean_norm(x):
2      s = 0.0
3      for x_i in x:
4          s += x_i**2
5      return sqrt(s)
```

In traditional low level operator-overloading AD one typically overloads the `__add__`, `__pow__`, and `sqrt` methods. This is also possible with pyadjoint. However in cases where we know how to symbolically compute the derivative of an operation, the recommended way to implement an overloaded version is to use this expression directly. This ensures that the tape does not grow unnecessarily large, which could cause performance problems as we will discuss in section 3.3.

Because this function returns only a single input, the above overloaded function code works perfectly with `target_function` replaced by `euclidean_norm`, and `TargetFunctionBlock` replaced by `EuclideanNormBlock`. Now it remains to implement the `TargetFunctionBlock` class. The main task of the block constructor is to store dependencies and save the necessary information needed for its adjoint model implementation. As seen above, the overloaded function handles the saving of the block output to the block instance.

```
class EuclideanNormBlock(Block):
    def __init__(self, x):
        Block.__init__(self)
        # Store the block variable as a dependency
        self.add_dependency(x.block_variable)
```

The attribute `block_variable` of the `OverloadedType` instance `x` stores the current block variable. If `x` was not an overloaded type, but instead a list-like structure of overloaded floats, then one would need to iterate over each entry of `x` and call `self.add_dependency` on each entry's block variable. For this example however, we assume that `x` is an `OverloadedType`.

In the case of `euclidean_norm` (denoted f in the following) the gradient consists of the expressions

$$\frac{\partial f(x)}{\partial x_i} = \frac{x_i}{(\sum_{j=1}^{n} x_j^2)^{\frac{1}{2}}} \tag{3.1}$$

for each $i = 1, 2, ..., n$. These partial derivatives are continuous for $x \neq 0$. Hence, the adjoint model of f as defined in 2.6 is

$$J^*_{f,x}(\delta y) = \frac{x \delta y}{|x|}$$

where $x \in \mathbb{R}^n \backslash \{0\}$ and $\delta y \in \mathbb{R}$.

This leads to the following implementation of the `evaluate_adj` method

```
def evaluate_adj(self):
    # Adjoint input
    delta_y = self.get_outputs()[0].adj_value
    # Forward output
    y = self.get_outputs()[0].saved_output
    # Forward input
    x = self.get_dependencies()[0].saved_output

    output = x/y * delta_y

    # Write the adjoint output
    self.get_dependencies()[0].add_adj_output(output)
```

The `saved_output` property of the block variables give access to the copy of the variable at the time of input/output. Note that the input x and the adjoint input δy are the only necessary inputs to the adjoint model. However, due to the structure of the block variables being both inputs and outputs of different blocks, the forward output is readily available and should be used for efficiency when needed.

3.2 User interface

We now present the key functions of the user interface. If pyadjoint has been correctly and fully applied to any python package, then the user should have access to all of the features below using the data types of the package for which we applied pyadjoint.

3.2.1 Computing gradients

The user interface includes several key functions and classes. To be able to compute any derivatives at all, one must initialise `Control` instances with variables. If `x` is an overloaded type and the derivative of a function with respect to `x` is the desired output, then one must define `control = Control(a)`. The need for a control class originates from the problem of multiple versions. If `x` is mutable and several versions of it is used in the forward computations, then saying that one wants the derivative with respect to `x` is ambiguous. Because pyadjoint itself has a unique way to identify versions through block variables, the control class is a very simple wrapper that only needs to store the relevant block variable.

Once one or several control instances have been initialised, computing a gradient can be done by a call to `compute_gradient(y, [x_1, x_2])`, where `x_1` and `x_2` must be control instances and `y` is the output for which we wish to compute the gradient. Another way of computing derivatives are through the `ReducedFunctional` class. This class takes in an overloaded `float` type, together with a list of one or more control instances. After a `ReducedFunctional` instance has been initialised, it provides methods such as computing gradients and recomputing the functional with new input values. In addition, a method provides a way to reduce the tape to only contain the necessary blocks for recomputing and differentiating the reduced functional. Lastly, the reduced functional interface is the main interface for using the optimization framework that is packaged with pyadjoint (see section 3.2.3).

3.2.2 Verification

An important part of developing software is testing. To test AD implementations one could implement analytical derivatives and test against them. However, this limits the range of problems to those where you have an analytical derivative and can be cumbersome. In particular, it can be very useful to verify that the model is correct before initiating a gradient-based optimization algorithm.

In pyadjoint the main tool for verifying that an implementation is correct is the `taylor_test` function. This is a test that verifies the convergence of the first order Taylor expansions remainder. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function. Given some initial point $x \in \mathbb{R}^n$ and some perturbation direction $\delta x \in \mathbb{R}^n$ then

$$|f(x + \epsilon\delta x) - f(x)| = \mathcal{O}(|\epsilon|), \quad \text{as } \epsilon \to 0$$

and

$$|f(x + \epsilon\delta x) - f(x) - \epsilon\nabla f \cdot \delta x| = \mathcal{O}(|\epsilon|^2), \quad \text{as } \epsilon \to 0$$

This allows for rigorous and efficient verification of the computed gradients.

3.2.3 Optimization

Because gradient-based optimization is a common application for AD generated derivatives, pyadjoint provides an optimisation framework that can be used

directly with the `ReducedFunctional`. The framework is essentially the exact same as was found in previous iterations of dolfin-adjoint [10] [12]. In essence it provides a wrapper for scipy [25] optimization functions. The author did minimal changes to the implementation of the optimization framework found in dolfin-adjoint. Most of the changes done stem from the need to move DOLFIN-specific [30] code outside of pyadjoint. This resulted in quite a few type specific abstract methods in the `OverloadedType` class. Which are then necessary to implement for the type to be compatible with the packaged optimization framework.

3.3 Limitations

As mentioned earlier, pyadjoint does not provide much value out of the box. The only included overloaded type is the overloaded `float`. Importing modules that enable more mathematical functions such as numpy [39] and the standard python library math, and using these imported functions on the overloaded `float` is often not going to work. In general, functions that use C implementations go outside the scope of what can be annotated automatically by pyadjoint. The Python interface for these C implementations must be overloaded and an adjoint model implemented for the whole underlying C computations.

Furthermore, although Python is an easy to use and highly flexible language, it can be painfully slow at performing mathematical operations. This is why most scientific computing python packages use some form of C/C++ implementation for the computationally expensive algorithms. Examples include numpy [39] and TensorFlow [1], as well as the emergence of tools such as Cython [6] that enables the compilation of (modified) Python code to C, which can be imported in a Python environment. Because of this, pyadjoint is not suitable for performing AD on large computational graphs. Instead, pyadjoint is most useful when the problems take a higher level of abstraction so that most of the computations are covered by a few number of blocks on the tape.

In relation to this, pyadjoint has to work with higher level data types such as vectors to be effective. Vector data types are most often mutable, and so pyadjoint requires a copy of them to work. Additionally, pyadjoint is by default quite aggressive in its copying. Every block variable has a unique copy of its `OverloadedType` instance. For large scale problems this can lead to issues with too high memory demands. This can be mitigated by using a checkpointing scheme in which only some copies are stored and others are recomputed once needed, starting from the last copy [15]. These algorithms aim to find the optimal balance between computational and memory expenses.

3.4 Summary

In this chapter we have presented a generic AD framework which can be applied to python software packages. The framework comes packed with tools that are often useful when working with derivatives, such as the optimization framework and verification methods. There are still room for improvement considering the current limitations of the framework.

Chapter 4

Applying pyadjoint to FEniCS

The aim of this chapter is to discuss how we apply pyadjoint to FEniCS [29]. The resulting implementation is named *dolfin-adjoint*. dolfin-adjoint is open source and available on bitbucket `https://bitbucket.org/dolfin-adjoint/pyadjoint/src`, or from the website `www.dolfin-adjoint.org`.

4.1 Brief introduction to FEniCS

FEniCS is a collection of software that enables automatic solving of partial differential equations using the finite element method [29]. DOLFIN [30] provides the main programming interface of FEniCS. The user defines the model problem as discrete variational forms through the domain-specific language UFL [3]. Both variational forms and functionals can be programmed in a notation similar to mathematical notation. These UFL forms are then compiled by a form compiler such as FFC [27], generating optimised low-level code for evaluating the variational forms on local elements. This code is then used by DOLFIN to assemble and solve the global system, making use of third-party libraries for the linear algebra functionality.

To better understand how we use UFL it can be useful to introduce some of the mathematical concepts behind UFL forms. Given a set $\{V_i\}_{i=1}^{\rho}$ of function spaces, we call the map

$$\begin{aligned} &a : V_1 \times V_2 \times \cdots \times V_\rho \to \mathbb{R} \\ &(v_1, v_2, ..., v_\rho) \mapsto a(v_1, v_2, ..., v_\rho) \end{aligned}$$

a *multilinear form*, given that a is linear in each v_i for all $i = 1, 2, ..., \rho$. We call the $\{v_i\}_{i=1}^{\rho}$ the arguments of a. The amount of arguments, ρ, is called the *arity* of a. For ρ equal to 0, 1, or 2 we call a a *functional*, *linear form*, or *bilinear form* respectively.

Additionally, we view a as parameterized by a set of coefficients $\{w_i\}_{i=1}^{k}$. Let $\{W_i\}_{i=1}^{k}$ be the coefficient function spaces, i.e. $w_i \in W_i$ for every $i = 1, 2, ..., k$. Then we may view a as a mapping from the product of the coefficient function spaces and argument function spaces

$$\begin{aligned} &a : W_1 \times W_2 \times \cdots \times W_k \times V_1 \times V_2 \times \cdots \times V_\rho \to \mathbb{R} \\ &(w_1, w_2, ..., w_k, v_1, v_2, ..., v_\rho) \mapsto a(w_1, w_2, ..., w_k; v_1, v_2, ..., v_\rho) \end{aligned}$$

In contrast to the arguments, the coefficients can be nonlinear in a. In general, a UFL form is defined by `integrand*dx`, where `dx` is the integration measure, signifying that the `integrand` is to be integrated. This UFL form then represents the multilinear form that has been parameterized by the coefficients appearing in the `integrand`. In other words, `a = integrand*dx` can be thought of as some multilinear form $a(w_1, w_2, ..., w_k; \cdot)$.

4.2 Annotation

As pyadjoint lets one freely decide at which level of abstraction to implement the AD types, we must first identify which operations and which data types should be overloaded and annotated. We consider that any model problem defined in FEniCS involves two main ingredients: partial differential equations and functionals. It is therefore reasonable to first implement pyadjoint annotation for the functions directly related to these operations. `solve` is a DOLFIN function that solves a variational problem or a linear system. While `assemble` can be used to evaluate a functional. Both of these functions accept UFL forms, and as we will see, the UFL forms and the algorithms provided for them, will be the key to an efficient implementation. Thus, we overload these functions and aim to store the UFL forms in their respective blocks. As usual we use the same overloading signature for these functions as described in section 3.1.2.

4.2.1 DOLFIN overloaded types

UFL forms consist of what is called coefficients. In DOLFIN the coefficients are implemented as the types `Function`, `Constant`, and `Expression`. A `Function` is defined on a finite element space, with its underlying data type being a k-length vector, where k is the number of basis functions spanning the finite element vector space, uniquely representing the function in the finite element space. The `Constant` is considered a constant-valued function. It can be scalar, vecor-valued or a tensor-valued function. Thus, the underlying vector is just in the codomain of the constant-valued function. In contrast, an `Expression` can not be represented as a vector before it has been evaluated. In our implementation we regard expressions as generic functions that can take on whatever suitable function space when needed. This is done by interpolation of the expression into the target function space.

We create overloaded types for `Function`, and `Constant`. Because `Expression` and `DirichletBC`, which is used for applying boundary conditions to systems and vectors, do not have an underlying data type, neither can be used as a control variable directly. However, they can be parameterized by functions or constants which in turn can be used as controls. Thus, Expressions and Dirichlet boundary conditions should still be annotated, and are therefore overloaded.

4.2.2 DOLFIN overloaded functions

Apart from the already presented `solve` and `assemble`, there are a few other important functions that need overloading. For example one can project an expression onto a finite element function space using the function `project`. This is essentially just solving a linear variational problem. Thus, we can regard this as a special `solve` call, and use the same block for `project`. The

`interpolate` function is also overloaded, but is not annotated. The purpose of overloading the interpolation function is to ensure that the returned function is an overloaded type instance. Hence, the most essential blocks to implement are the blocks for `solve` and `assemble`. There are other blocks, but they will not be mentioned in this thesis.

4.3 Deriving the discrete tangent linear and adjoint equations

In order to implement the block representing a `solve`, we first must be able to derive the equations described in section 2.4.2. UFL provides powerful form operators that allows the user to produce new forms from existing forms. This is implemented through efficient algorithms exploiting the symbolic representation of the UFL forms. For deriving the discrete tangent linear and adjoint equations, the most important form operators are `action`, `adjoint`, `derivative`, and `replace`. Their usage is summarised in Table 4.1. The perhaps most powerful of these is the `derivative` function. It differentiates a form with respect to any coefficient. This can either be in a certain direction, introducing a new coefficient, or it can be done in an arbitrary direction, introducing a new argument and resulting in a new form with one greater arity. Hence, if we have the linear form $F(u, m; v)$ representing our PDE, we can obtain the terms in the tangent linear equation (2.17). The left hand side of in the tangent linear equation (2.17) is the derivative in the direction $\mu_{\delta m}$. In other words, this direction is the unknown. Thus, it makes sense to differentiate F with respect to u in an arbitrary direction $\mu_{\delta m}$. For $\partial F/\partial m$ on the other hand, the direction is given as δm. Thus we end up with the problem of finding $\mu_{\delta m}$ such that

$$\partial_u F(u, m; \mu_{\delta m}, v) = \partial_m F(u, m, \delta m; v), \quad \forall v \in V_h'$$

which is just the discrete variational formulation of a linear PDE.

One can obtain the left hand side of the adjoint equation (2.18) in almost the exact same way. The arbitrary direction is now instead thought of as the test function v, as we will later apply the adjoint form operator. Thus the arguments can be thought of as in reverse order, $F(u, m; \lambda_{\delta u}) \mapsto \partial_u F(u, m; v, \lambda_{\delta u})$. When applying the `adjoint` form operator the result is $\partial_u F^*(u, m; \lambda_{\delta u}, v) = \partial_u F(u, m; v, \lambda_{\delta u})$, the arguments just swaps places in the form. The right hand side of (2.18) is δu^*, which is the adjoint input. Finally, we apply boundary conditions as described in section 2.4.3, assuming that m does not parameterize the Dirichlet boundary condition. The tangent linear equation and adjoint equations can be derived and solved in DOLFIN as follows.

```
1  # Homogenize bcs
2  bcs = [bc.homogenize() for bc in bcs]
3
4  # Tangent linear equation
5  mu = TrialFunction(u.function_space())
6  dF_u = derivative(F, u, mu)
7  dF_m = derivative(F, m, delta_m)
8  solve(dF_u == dF_m, mu_sol, bcs)
9
10 # Adjoint equation
```

Mathematical notation	UFL notation	Description
$a \mapsto a^*$	`a_star = adjoint(a)`	Derive adjoint form of bilinear form a
$F(f;\cdot) \mapsto F(g;\cdot)$	`G = replace(F, {f:g})`	Replace coefficient `f` with `g` in `F`
$F(;\cdot) \mapsto F(f;\cdot)$	`M = action(F, f)`	Replace argument function 1 in `F` by `f`
$F(f;\cdot) \mapsto \partial_f F(f;\cdot)[v]$	`dF = derivative(F, f, v)`	Differentiate `F` w.r.t `f` in direction `v`

Table 4.1: The primarily used UFL form operators in dolfin-adjoint. For a multilinear form F of arbitrary arity. Extracted from Table XII on page 16 in [3]

```
11 v = TrialFunction(u.function_space())
12 dF_u = derivative(F, u, v)
13 dF_u_adj = adjoint(dF_u)
14 solve(dF_u_adj == adj_input, lmbd, bcs)
```

4.4 Dirichlet boundary condition control

Given a PDE defined through the linear UFL form $F(u, m; v)$ the `derivative` and `adjoint` functions can be used to derive the left hand side of the adjoint equation. However, prior to assembly the Dirichlet boundary conditions cannot be applied. This means that we have two options to obtain the system in equation (2.21). The first option is to perform `adjoint`, assemble and then zero out the relevant columns leaving 1 on the diagonal element of those columns. While the second option is to assemble, then apply boundary conditions and lastly transpose the matrix. Both of these methods go outside of the DOLFIN interface, and we wish to avoid them. Thus, we apply the homogeneous Dirichlet boundary conditions on the adjoint assembled system, restricting the solve to only finding the interior adjoint solution such as in section 2.4.3.

Using the notation from 2.4.3, the adjoint system assembled from the UFL forms is

$$G^*\lambda = \delta u^*$$
$$\begin{pmatrix} (G^{00})^* & (G^{g0})^* \\ (G^{0g})^* & (G^{gg})^* \end{pmatrix} \begin{pmatrix} \lambda^0 \\ \lambda^g \end{pmatrix} = \begin{pmatrix} (\delta u^0)^* \\ (\delta u^g)^* \end{pmatrix}$$

where λ^g is the adjoint solution at the boundary where Dirichlet boundary conditions apply. Applying homogeneous Dirichlet boundary conditions yields the same system as (3.1) because $\lambda^g = 0$ which in turn makes $(G^{g0})^*\lambda^g = 0$.

To enable strong boundary condition controls, the adjoint solution at the boundary λ^g is necessary. Following the description in section 2.4.3, λ^g is found by

$$\lambda_{\text{bdy}} = \begin{pmatrix} c \\ \lambda^g \end{pmatrix} = \begin{pmatrix} (\delta u^0)^* \\ (\delta u^g)^* \end{pmatrix} - \begin{pmatrix} (G^{00})^* & (G^{g0})^* \\ (G^{0g})^* & (G^{gg})^* \end{pmatrix} \begin{pmatrix} \lambda^0 \\ 0 \end{pmatrix}$$

where c represents what is computed for the interior, but is otherwise ignored. Even though we are only interested in computing the lower half of λ_{bdy}, there

is no practical way of extracting only the relevant block matrices in DOLFIN. Regardless, the actual implementation is rather compact as illustrated below.

```
# Compute on boundary,
# adj_input must be copied to avoid bc.apply.
adj_bdy = adj_input_copy - assemble(action(dF_u_adj, lmbd))
```

The second ingredient for implementing Dirichlet boundary condition control is the computation of $\partial F/\partial m$ when the Dirichlet boundary condition is parameterized by m. Because the strong boundary condition does not appear as a coefficient in F, it is not possible to use `derivative` to automatically derive the correct values. Instead, the implementation is based on the realization that $\partial F/\partial m$ is a matrix with entries 1 on the diagonal entries of rows corresponding to the Dirichlet boundary condition and with entries 0 elsewhere. This can be seen by noticing that the Dirichlet boundary function $m = g$ only modifies the right hand side of the system

$$A(u)u = b(u, m)$$

and that

$$F(u, m) = A(u)u - b(u, m) \implies \frac{\partial F}{\partial m} = -\frac{\partial b}{\partial m} = -\begin{pmatrix} 0 & 0 \\ 0 & I \end{pmatrix}$$

Finally, using (2.19) one can compute the adjoint model with respect to the Dirichlet boundary condition as

$$J_{u,m}(\delta u) = -\frac{\partial F}{\partial m}^* \lambda_{\delta u} = \begin{pmatrix} 0 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \lambda^0_{\delta u} \\ \lambda^g_{\delta u} \end{pmatrix} = \begin{pmatrix} 0 \\ \lambda^g_{\delta u} \end{pmatrix}$$

4.5 Parallel support

dolfin-adjoint inherits its parallel support from FEniCS. When running a FEniCS model in parallel, the underlying vectors and system matrices are automatically distributed among the parallel processes. Because dolfin-adjoint uses the high level abstraction present in the UFL forms to derive the adjoint equations, the resulting adjoint equations are perfectly valid UFL forms that can be solved using the DOLFIN interface. Thus, the assembly and solving of the adjoint equations are automatically run in parallel given that the forward computations are run in parallel.

4.6 Limitations

The most glaring limitation of the dolfin-adjoint implementation is that access to the underlying vectors of a `Function` is limited. If a function's vector is modified by direct access to the memory location, the current implementation does not annotate this operation. This can possibly lead to the adjoint computations being incorrect. A possible solution to this problem is to overload the access and modification functions such that they are annotated. This would work by overloading `GenericVector.__setitem__`, and annotating a modification as a vector assignment to another vector. From this one could even explore annotating the extraction of single vector entries as an operation going

from a vector to a scalar, where the scalar would be an overloaded float as included in pyadjoint. However, annotating direct vector access like this can easily become problematic. Imagine the case where a user iterates through a large vector to assign values pointwise. Then the result would be k extra blocks on the tape, where k is the length of the vector. Because pyadjoint can be inefficient for very large tapes, the benefit of implementing such a solution might not outweigh the drawback of the possible additional overhead.

4.7 Summary

In this chapter we presented the application of pyadjoint to FEniCS. This resulted in the new iteration of dolfin-adjoint, which derives the adjoint equations through the high level of abstraction present in UFL forms. This results in equations that are perfectly valid FEniCS code and can utilize the assembly and solving interface of FEniCS. This leads to the natural support for solving the adjoint equations in parallel. Furthermore, the adjoint solution on the boundary is computed by using back substitution after having computed the adjoint solution on the interior. This was then used to implement support for strong Dirichlet boundary conditions controls.

Chapter 5

Examples

In this chapter we provide examples of using the library developed in the previous chapter. Specifically, we show the use of strong boundary condition controls in section 5.1, verifying that the gradient is correct by Taylor remainders tests. We also provide a benchmark of the problem to test runtime performance. In section 5.2 we use test dolfin-adjoint with an inverse problem, using the pyadjoint-included optimization functions in parallel with 2 CPU cores. This illustrates that the optimization user interface is working in parallel.

5.1 Dirichlet boundary condition control

In this example we consider Stokes equations with Dirichlet boundary conditions, and wish to compute the sensitivity of a functional with respect to the boundary condition on a sub-boundary. The aim is to illustrate that gradients with Dirichlet boundary condition control can be computed and that the computations are correct as verified by the Taylor remainders test. Lastly, we benchmark the computational time of the gradient computations, and compare it to the forward computations.

5.1.1 Problem definition

We consider the domain Ω as illustrated in figure 5.1 below. This is a rectangle with a cut-out circle in the interior. The long edges of the rectangle are defined as the boundaries $\partial\Omega_{\text{walls}}$, while the left edge and the right edge of the rectangle are the inflow boundary $\partial\Omega_{\text{in}}$ and outflow boundary $\partial\Omega_{\text{out}}$ respectively. The boundary on the circle is denoted $\partial\Omega_{\text{circle}}$, and is where we prescribe our Dirichlet boundary condition control.

Consider the Stokes equations with Dirichlet boundary conditions,

$$\begin{aligned} -\nu\Delta u + \nabla p &= 0, && \text{in } \Omega \\ \nabla \cdot u &= 0, && \text{in } \Omega \\ u &= 0, && \text{on } \partial\Omega_{\text{walls}} \\ u &= f, && \text{on } \partial\Omega_{\text{in}} \\ u &= g, && \text{on } \partial\Omega_{\text{circle}} \\ p &= 0, && \text{on } \partial\Omega_{\text{out}} \end{aligned}$$

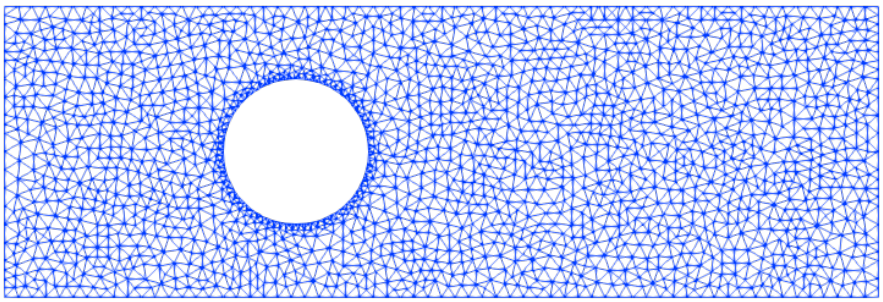


Figure 5.1: The domain Ω on which we will be solving the Stokes equations.

where $u : \Omega \to \mathbb{R}^2$ is the unknown velocity, $p : \Omega \to \mathbb{R}$ is the unknown pressure, $\nu \in \mathbb{R}$ is the viscosity. Where f and g are given Dirichlet boundary functions. The problem is to compute the sensitivity of the functional

$$J(u, m) = \int_\Omega \nabla u \cdot \nabla u \, dx$$

with respect to the boundary function g.

The variational formulation of this problem is: find $u \in H^1_{\text{bcs}}(\Omega)$ and $p \in L^2(\Omega)$ such that

$$a(u, p, v, q) = L(u, p, v, q), \quad \forall v \in H^1_0, q \in L^2(\Omega)$$

where

$$\begin{aligned} a(u, p, v, q) &= \nu \int_\Omega \nabla u : \nabla v - \int_\Omega p \nabla \cdot v \, dx - \int_\Omega q \nabla \cdot u \, dx \\ L(u, p, v, q) &= 0 \end{aligned}$$

Lastly, for the code below we will define the reduced functional $\hat{J}(g) = J(u(g))$, and compute the gradient $\nabla \hat{J}(g)$.

5.1.2 Implementation

We illustrate the minimalistic change of code needed to apply AD by showing a simplified code snippet of the implementation. Only three lines are needed for the adjoint specific code, one import statement and two lines to define the reduced functional and compute its derivative.

```
0  from dolfin import *
1  from dolfin_adjoint import *
2
3  bc = DirichletBC(W.sub(0), g)
4
5  a = (nu*inner(grad(u), grad(v))*dx
6       - inner(p, div(v))*dx
7       - inner(q, div(u))*dx
8       )
9  L = inner(Constant([0, 0]), v)*dx
10
```

ϵ	Residual	Order	Gradient Residual	Gradient order
$1 \cdot 10^{-2}$	7.3960		$3.2891 \cdot 10^{-1}$	
$5 \cdot 10^{-3}$	3.7800	1.0	$8.2499 \cdot 10^{-2}$	2.0
$2.5 \cdot 10^{-3}$	1.9106	1.0	$2.0667 \cdot 10^{-2}$	2.0
$1.25 \cdot 10^{-3}$	$9.6044 \cdot 10^{-1}$	1.0	$5.1723 \cdot 10^{-3}$	2.0

Table 5.1: The Taylor remainders without gradient information and with gradient information. The computed convergence orders are as expected.

```
11 A, b = assemble_system(a, L, bcs)
12 solve(A, s.vector(), b)
13
14 u, p = split(s)
15 J = assemble(inner(grad(u), grad(u))**2*dx)
16
17 # dolfin-adjoint specific code
18 Jhat = ReducedFunctional(J, Control(g))
19 dJdm = Jhat.derivative()
```

5.1.3 Verification

The implementation is verified using the taylor test as described in 3.2.2. The perturbation direction is randomly generated with uniform distribution $[-0.5, 0.5)$. The results are summarised in table 5.1. All the computed values are as expected, which indicates that the implementation is correct.

5.1.4 Benchmarking

We end the example with a benchmark of the implementation. Because the PDE is linear, computing the gradient involves solving the same amount of PDEs as in the forward computations. If a PDE solve is where the main computational expense originates from, then it makes sense that the computational time for the gradient computations is also dominated by the adjoint PDE solves. Because solving the forward PDEs and the adjoint PDEs both involve solving linear systems of the same size, the optimal ratio between forward computations and gradient computations should be close to 1. In the following benchmark we ran the forward and adjoint computations 5 times and produced an average runtime for both that is summarised in table 5.2. The results show a close to optimal performance in this example. However, it should be noted that the example essentially only involves three blocks, one for `solve`, one for `assemble`, and one for `DirichletBC`. Thus, the overhead caused by the inner workings of the AD framework is minimal.

Forward runtime (s)	17.1
Adjoint runtime (s)	18.0
Adjoint/Forward ratio	1.05

Table 5.2: The benchmarks of 5 runs of the forward and adjoint computations. The ratio is close to the optimal 1.0.

5.2 Brain diffusion inverse problem

We demonstrate the application of our implementations on a real world problem. The problem at hand is an inverse problem of identifying diffusion coefficients for the diffusion of contrast agents in the brain. Section 5.2.1 tries to give a brief background on the relevancy of this problem. We will not be handling the full problem, but instead will be going through some of the steps that are relevant for testing that the problem is correctly implemented and can be solved. This includes testing parallel functionality, correct convergence of the optimization from generated observations (with and without noise), and identification of regularisation parameters.

5.2.1 Background

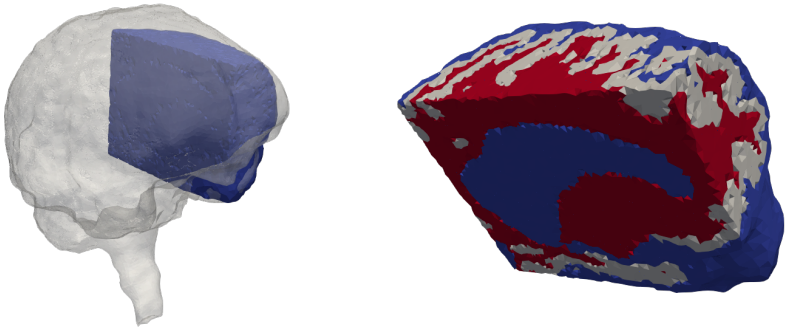
Understanding of how substances are delivered and cleared from and to the central nervous system (CNS) is an important research topic. The blood brain barrier is a big hurdle for delivering therapeutic drugs to the CNS as only lipophilic drugs are able to diffuse and pass into the brain [43]. For this reason, drugs administered intrathecally have been a topic of interest, with some showing promising results [11]. However, the mechanisms behind transport of these substances are not fully understood. The brain lacks lymphatic vessels in the parenchyma, but is dependent on clearance of waste substances such as Amyloid β. Indeed, a lack of clearing such substances is correlated with diseases such as Alzheimer's and multiple sclerosis [32].

In 2012 a study suggested a new brain-wide pathway for the clearance of these substances [21]. This hypothesised a clearance process driven by convective bulk flow of cerebral spinal fluid (CSF) through the interstitial space, which was termed the glympathic system. However, later modelling studies suggests that the transport of substances through the interstitial space cannot be explained by bulk flow alone [20] [24]. Instead, these simulation studies suggest that diffusion plays a more significant role in the transportation process.

In a current study it has been shown enrichments of CSF tracer in deeper brain white matter using the magnetic resonance imaging (MRI) contrast agent gadobutrol. Using data of contrast concentration from these MRI images, it is of interest to fit a model for diffusion to these observations to estimate apparent diffusion coefficients. Although we will be employing a diffusion model for CSF spaces, convection is a primary driver for the spreading of contrast agent in CSF [17]. Thus, we emphasize that we are not actually computing diffusion coefficient for CSF, but rather apparent diffusion coefficient.

5.2.2 Problem definition

The spatial domain Ω is a 3-dimensional region of a human brain. As seen in figure 5.2a this subset is a part of the left side of the brain. The mesh we will be working with is rather coarse, with only 22057 vertices. For comparison the mesh used for experimentation on real patient-specific data has around half a million vertices. Ω is divided into 3 subdomains, one for CSF, one for grey matter and one for white matter. The three subdomains are shown in different colours in figure 5.2b. The boundary Γ is defined as the outer boundary of the CSF subdomain. In other words, if $\partial\Omega$ is the boundary of the whole spatial domain and $\partial\Omega_{CSF}$ the boundary of the CSF subdomain, then $\Gamma = \partial\Omega \cap \partial\Omega_{CSF}$. This is indicated by the blue area in figure 5.2b.



(a) (b)

Figure 5.2: (a) shows an illustration of the subset of the brain we are using as domain. While (b) shows the different subdomains. The blue area in figure (b) is the CSF, the grey area is the grey matter, and the red area is the white matter. We will impose Dirichlet boundary conditions to all of the blue area on the outside of figure (b).

As mentioned, we use the diffusion equation as a model for the contrast.

$$\frac{\partial u}{\partial t} - \nabla \cdot D\,\nabla u = 0, \quad \text{in } \Omega \times (0,T), \tag{5.1}$$

$$u = g, \quad \text{on } \Gamma \times (0,T) \tag{5.2}$$

$$u = u_0, \quad \text{in } \Omega \times \{0\} \tag{5.3}$$

where $u : \Omega \times (0,T) \to \mathbb{R}$ is the solution yielding the concentration of the contrast agent in the spatial and temporal domain. D is the diffusion coefficient, which is a different scalar constant for each subdomain. Furthermore, $g : \Gamma \times (0,T) \to \mathbb{R}$ is the Dirichlet boundary conditions and $u_0 : \Omega \to \mathbb{R}$ is the known initial condition. We are interested in finding the diffusion coefficients D and boundary conditions g that best fit the observations $\{d_i\}_{i=1}^{k_d}$, where k_d is the amount of observations. In addition to the PDE-constraint above, we

also impose bounds on the controls

$$0 \leq g \leq M_g$$
$$0 \leq D \leq M_D$$

for some given positive values $M_g, M_D \in \mathbb{R}$.

The temporal domain will be uniformly discretized using a Backward Euler scheme, with step size Δt and k time steps:

$$\Delta t = \frac{T}{k},$$
$$t_i = i\Delta t, \quad \text{for } i = 0, 1, ..., k$$

As our observations are not so neatly divided in time, having more frequent observations early on and less later on, we define a set $\mathcal{I}_d$ of the time indices which fall closest to an observation. With $s : \mathcal{I}_d \to \mathbb{N}$ mapping these time indices to the corresponding observation indices. Thus our functional of interest becomes

$$J(u) = \sum_{i \in \mathcal{I}_d} \left\| u(\cdot, t_i) - d_{s(i)}(\cdot) \right\|^2_{L^2(\Omega)}$$

Even if the forward problem yields a unique solution for each set of parameters D and g, the inverse problem is usually not unique. If the problem has no solution, or multiple solutions we call it ill-posed. To mitigate this we add regularisation terms which should ensure that the problem is well-posed and that the boundary solution has some amount of smoothness in time.

$$\mathcal{R}(g) = \frac{\alpha}{2} \int_0^T \|g\|^2_{L^2(\Gamma)} \, dt + \frac{\beta}{2} \int_0^T \|\dot{g}\|^2_{L^2(\Gamma)} \, dt$$

with $\dot{g} = \partial g / \partial t$, for some constant regularisation parameters α and β. The problem then becomes

$$\min_{u,D,g} J(u) + \mathcal{R}(g) \text{ subject to (5.1)} - \text{(5.3)}$$

To employ the finite element method, we derive the variational form. After discretizing in time by a Backward Euler scheme, our problem formulation becomes: for $i = 1, 2, ..., k$ find $u_i \in V_g$ such that

$$a(D; u_i, v) = L(u_{i-1}; v), \quad \forall v \in V_0 \tag{5.4}$$

with

$$a(D; u_i, v) = \int_\Omega u_i v \, dx + \Delta t \int_\Omega D \nabla u_i \cdot \nabla v \, dx \tag{5.5}$$
$$L(u_{i-1}; v) = \int_\Omega u_{i-1} v \, dx \tag{5.6}$$

where u_i denotes $u(\cdot, t_i)$. We will be using first-degree Lagrange elements in the finite element discretization.

5.2.3 Implementing the model

We will now implement the model by defining a function that evaluates $J(u) + \mathcal{R}(g)$ given some D and g. After the `mesh` has been loaded, both the `subdomains` and `boundaries` have been marked, and the function space `V` has been defined, we can define the function `forward_problem`. `forward_problem` takes three arguments: the diffusion coefficient `D`, the list of boundary functions `g_list` and the observation time points `tau`. Because we want to stop after the last observation has been used, we choose the last observation time point as T. Furthermore, we have one boundary condition function for every time step, i.e. $\{g_i\}_{i=1}^k$ with $g_i(x) = g(x, t_i)$, so we determine Δt from the last element of `tau` and the length of the list `g_list`.

```
def forward_problem(D, g_list, tau, alpha=0.0, beta=0.0):
    """Compute the forward problem and return J + R.

    Args:
        D (dict or list): dict or list of diffusion coefficients,
            with keys/index being subdomain id corresponding to
                                                that coefficient.
        g_list (list): list of all boundary conditions.
            Length determines amount of timesteps
        tau (list): list of all observation time points.
            Last time point determines stop time (T) for
                                                simulation.
        alpha (float, optional): Regularisation parameter of
                                                boundary condition g
        beta (float, optional): Regularisation parameter of time
                                                derivative of g

    Returns:
        float: the resulting objective functional with
                                                regularisation: J + R.

    """
```

As our diffusion coefficient is defined as a constant on each subdomain, we split the second integral in (5.5) into 3 integrals, one for each subdomain. The subdomains are marked as $1, 2$, and 3, for CSF, grey matter, and white matter respectively. From this we express the bilinear and linear form (5.5)-(5.6) as UFL forms,

```
a = u * v * dx + sum([dt * D[j] * inner(grad(v), grad(u)) * dx(j)
                                    for j in range(1, 4)])
L = U_prev * v * dx
```

Then after pre-assembling the matrix for the bilinear form a, we must construct a loop for solving the problem forwards in time. Inside we both solve the problem and construct the functional with regularisation. The loop stops once every observation has been used, which should coincide with k. The time integrals are computed using the trapezoidal rule, and the time derivative of g is computed by finite difference. A simplified snippet showing the construction of the functional can be seen below. In the code `J` denotes the sum $J(u)+\mathcal{R}(g)$.

```
J = 0
while next_tau < len(tau):
    # Advance in time, assign to U_prev
    # prescribe boundary conditions for this timestep,
```

```
    # set g_prev to g,
    # solve the problem
    ...

    if abs(t - tau[next_tau]) < abs(t + dt - tau[next_tau]):
        d = ...  # Read observation
        J += assemble((U - d) ** 2 * dx)
        next_tau += 1  # Move on to next observation

    # Choose time integral weights
    if t <= dt or next_tau >= len(tau):
        # If endpoints use 0.5 weight
        weight = 0.5
    else:
        # Otherwise 1.0 weight
        weight = 1.0

    # Add regularisation
    J += alpha/2*weight*dt*assemble(g**2*ds(1))
    if current_g_index > 1:
        J += beta/2*weight*dt*assemble(((g - g_prev)/dt)**2*ds(1))

return J
```

Finally, we can compute our forward problem by something similar to the following.

```
D = {1: Constant(1000.0), 2: Constant(1.0), 3: Constant(2.0)}
k = 20  # Time steps
g = [Function(V) for i in range(k)]
tau = [0.1, 0.3, 0.7, 1.3, 2.0]  # Observation time points
J = forward_problem(D, g, tau)
```

To run the optimization, we add the pyadjoint specific code. The optimization will be run in parallel.

```
# Define the control variables
controls = ([Control(D[i]) for i in range(1, 4)]
             + [Control(g_i) for g_i in g])
# Define the reduced functional
Jhat = ReducedFunctional(J, controls)

# Define the bounds
# For the diffusion coefficients
lb = [0, 0, 0]
ub = [M_D, M_D, M_D]

# For the boundary functions
for i in range(3, len(ctrls)):
    lb.append(0.0)
    ub.append(M_g)

# Run the optimization
optimal_controls = minimize(Jhat, bounds = (lb, ub))
```

5.2.4 Generating observations

We generate observations by choosing some diffusion coefficients and boundary conditions and computing the forward problem. The chosen diffusion coefficients are $1000.0, 2.0$, and 1.0 for CSF, grey matter, and white matter respec-

ϵ	Residual	Order	Gradient Residual	Gradient order
$1 \cdot 10^{-2}$	3.7254		$1.3999 \cdot 10^{-2}$	
$5 \cdot 10^{-3}$	1.8592	1.0	$3.4877 \cdot 10^{-3}$	2.0
$2.5 \cdot 10^{-3}$	$9.2873 \cdot 10^{-1}$	1.0	$8.6584 \cdot 10^{-4}$	2.0
$1.25 \cdot 10^{-3}$	$4.6414 \cdot 10^{-1}$	1.0	$2.1342 \cdot 10^{-4}$	2.0

Table 5.3: The Taylor remainders without gradient information and with gradient information. The computed convergence orders are as expected. See section 3.2.2 for more information on Taylor remainders tests.

tively. The model is ran with $k = 20$ time steps with $T = 2.0$, making $\Delta t = 0.1$. Out of these 20 solutions, $k_d = 5$ are used as observations with increasing gaps to mimic the sparsity of real data. Specifically, the chosen observation time points are $\mathcal{I}_d = \{0.1, 0.3, 0.7, 1.3, 2.0\}$. The Dirichlet boundary function g were chosen to be $g(x, t) = (2 - t)t + 0.5$, reaching its peak at $t = 1.0$.

5.2.5 Verification

After importing `dolfin_adjoint`, we wish to verify that our model is correctly implemented and compatible with our adjoint software. This will be achieved by Taylor remainders tests of the forward problem and gradient, as described in section 3.2.2. All initial control values and the perturbation directions are randomly chosen using pointwise uniform distribution with values deviating by up to 50% from the exact solution. Regularisation is enabled with $\alpha = \beta = 10^{-4}$. The results of the taylor tests are summarised in table 5.3. The tests give expected orders of convergence both for serial and parallel, which indicate that the model have been correctly implemented.

5.2.6 Finding regularisation parameters

Now the aim is to find suitable regularisation parameters α and β. To this end, we test different parameter values and compare how fast the optimization algorithm converges and if the converged control values are close to the actual optimal values. For this we employ the quasi-newton optimization algorithm Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) [7] implemented in scipy [25]. The stopping criteria is

$$\frac{f^k - f^{k-1}}{\max(|f^k|, |f^{k-1}|, 1)} \leq 10^{-6}$$

where f^k is the value of objective functional $J + \mathcal{R}$ at iteration k. Additionally, the optimization will stop after 100 iterations if 5.2.6 is not satisfied.

We performed the optimization with regularisation parameters at 10^{-4}, 10^{-2}, and 1. The initial guess was $350, 0.8$, and 0.8 for the diffusion coefficients at CSF, grey matter and white matter respectively, with the initial boundary condition guess at zero. To measure the error of the resulting optimal solution

α \ β	1	10^{-2}	10^{-4}
1	$(547.895, 2.149, 0.984)$	$(519.838, 2.357, 0.964)$	$(535.911, 2.508, 0.955)$
10^{-2}	$(550.550, 2.044, 0.963)$	$(448.721, 1.883, 0.953)$	$(436.600, 1.887, 0.993)$
10^{-4}	$(525.742, 2.103, 0.967)$	$(456.131, 1.868, 0.984)$	$(453.746, 1.846, 0.999)$

Table 5.4: The approximated diffusion coefficients for (CSF, grey matter, white matter). The correct values are $(1000.0, 2.0, 1.0)$, and we see relatively close approximations for both grey matter and white matter, while the diffusion coefficient for CSF converges much more slowly.

we use the relative error norm

$$\mathcal{E} = \frac{\|u - u_{\text{true}}\|_{L^2(\Omega\times(0,T])}}{\|u_{\text{true}}\|_{L^2(\Omega\times(0,T])}}$$

where $u_{\text{true}}(x,t)$ is the generated solution from which the observations are taken and u is the solution after optimization. The results are summarized in figure 5.3. None of the optimization runs reached the stopping criteria within 100 iterations, and thus the approximations are rather poor when compared to the true solution. Notice that for $\beta < 1$ we get oscillations in the norm $\|u\|_{L^2(\Omega)}$. This is caused by the boundary conditions oscillating in time, which stems from a too relaxed temporal smoothness parameter β. It is therefore reasonable to choose $\beta = 1$ and vary α. Tests reveal that $\alpha = 10^{-2}$ is the best candidate out of the four, yielding relative error $\mathcal{E} = 10\%$ after 100 iterations.

Table 5.4 shows the diffusion coefficients resulting from 100 iterations for each pair of regularisation parameters. The diffusion coefficients for the grey and white matter subdomains converge relatively fast. Meanwhile the CSF coefficient coupled with the boundary conditions make the optimization go really slow.

We test the use of a second optimization algorithm, namely a truncated Newton algorithm (TNC) [37] available in scipy. Interestingly, using this optimization algorithm on the results from the 100 iterations of L-BFGS, yields much more accurate CSF diffusion coefficients. For $\alpha = 10^{-2}$ and $\beta = 1$, at a maximum of 100 functional evaluations the estimated diffusion coefficients are $(995.245, 2.102, 1.012)$. By comparison, another 100 iterations of L-BFGS, which roughly equals 120 functional evaluations, the estimated diffusion coefficients are $(657.606, 2.113, 1.013)$. The results are summarised in figure 5.4. Observe that only using TNC results in a very poor convergence, but starting with L-BFGS and then using TNC gives the optimal results. It looks like TNC requires the boundary condition to not be so far from the optimum for it to converge. Furthermore, notice that the difference between CSF coefficient of 657.606 and 999.043 does not reflect much in the relative error.

5.2.7 Adding noise

We finish by testing the optimization with noise. To this end, we add pointwise Gaussian noise to the observations. The optimization is tested on Gaussian noise with standard deviation (SD) $\sigma = 0.1$ and $\sigma = 0.3$. We only test the

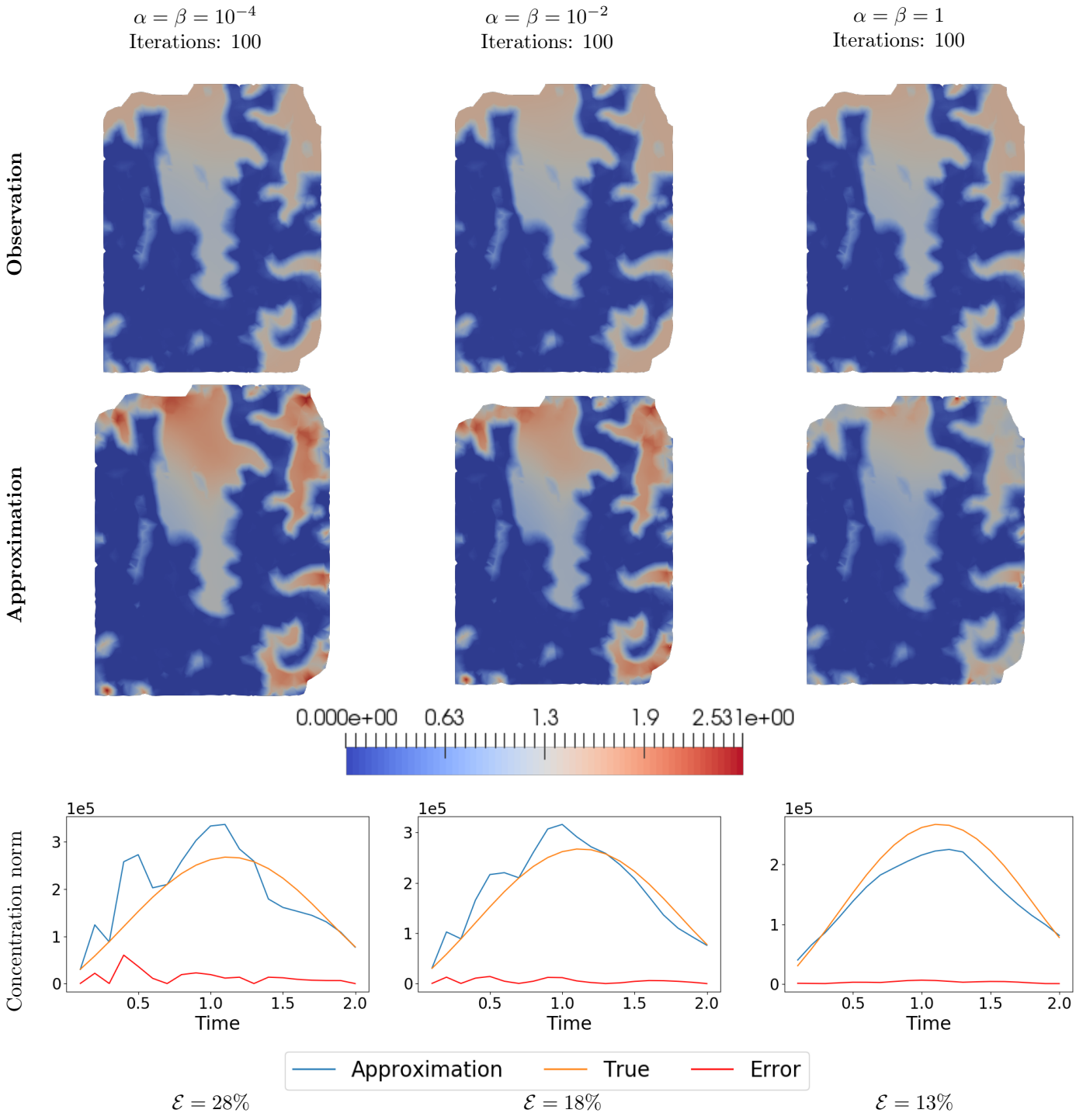


Figure 5.3: Results of different regularisation parameters. The top two rows feature cross-section snapshots at $t = 1.0$.

noise against the mix of 100 L-BFGS iterations followed by 100 TNC functional evaluations, with $\alpha = 10^{-2}$ and $\beta = 1$. The results are summarised in figure 5.5. Noise does not seem to matter extremely much, although for $\sigma = 0.3$ we did observe quite a large discrepancy in the approximate white matter coefficient and the true coefficient. Because the optimizations were all stopped before a convergence criteria was met, it is hard to predict if this discrepancy was caused by slower convergence or if the noise leads to a wrong optimal solution all together.

5.3 Summary

We presented two examples that illustrated the application of dolfin-adjoint. Through the Stokes example in section 5.1, the gradient of a reduced functional with respect to a Dirichlet boundary condition control was computed with minor modifications to the code. The computed gradients were then verified, indicating that AD with Dirichlet boundary condition controls are correctly implemented. The example was also benchmarked, showing close to optimal runtime for the adjoint computations. However, benchmarks have not been performed on more complex and large-scale problems, so it can not be assumed that the implementation has optimal performance in general.

In section 5.2 the use of the `ReducedFunctional` with the optimization framework was illustrated on an inverse problem with generated observations. The optimization was run in parallel and showed ability to get close to an optimal solution. However, due to time constraints, the program was not able to reach a convergence criterion. Moreover, the example verified the use of multiple control values consisting of both time dependent Dirichlet boundary conditions and constant-valued functions.

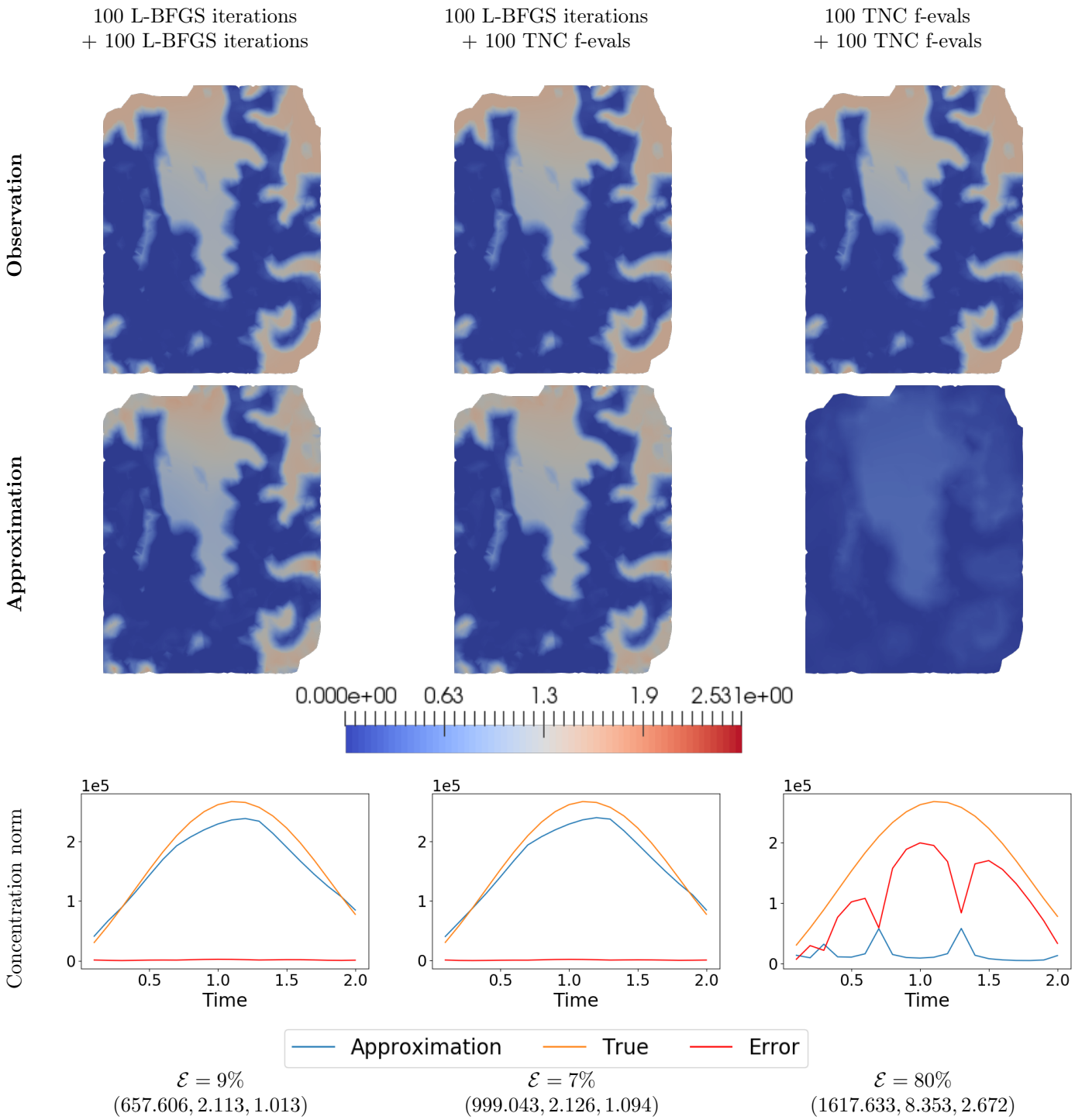


Figure 5.4: Results of different combinations of optimization algorithms. All three cases are run with $\alpha = 10^{-2}$ and $\beta = 1$. At the bottom we also include the estimated diffusion coefficients in the order (CSF, grey matter, white matter). The top two rows feature cross-section snapshots at $t = 1.0$.

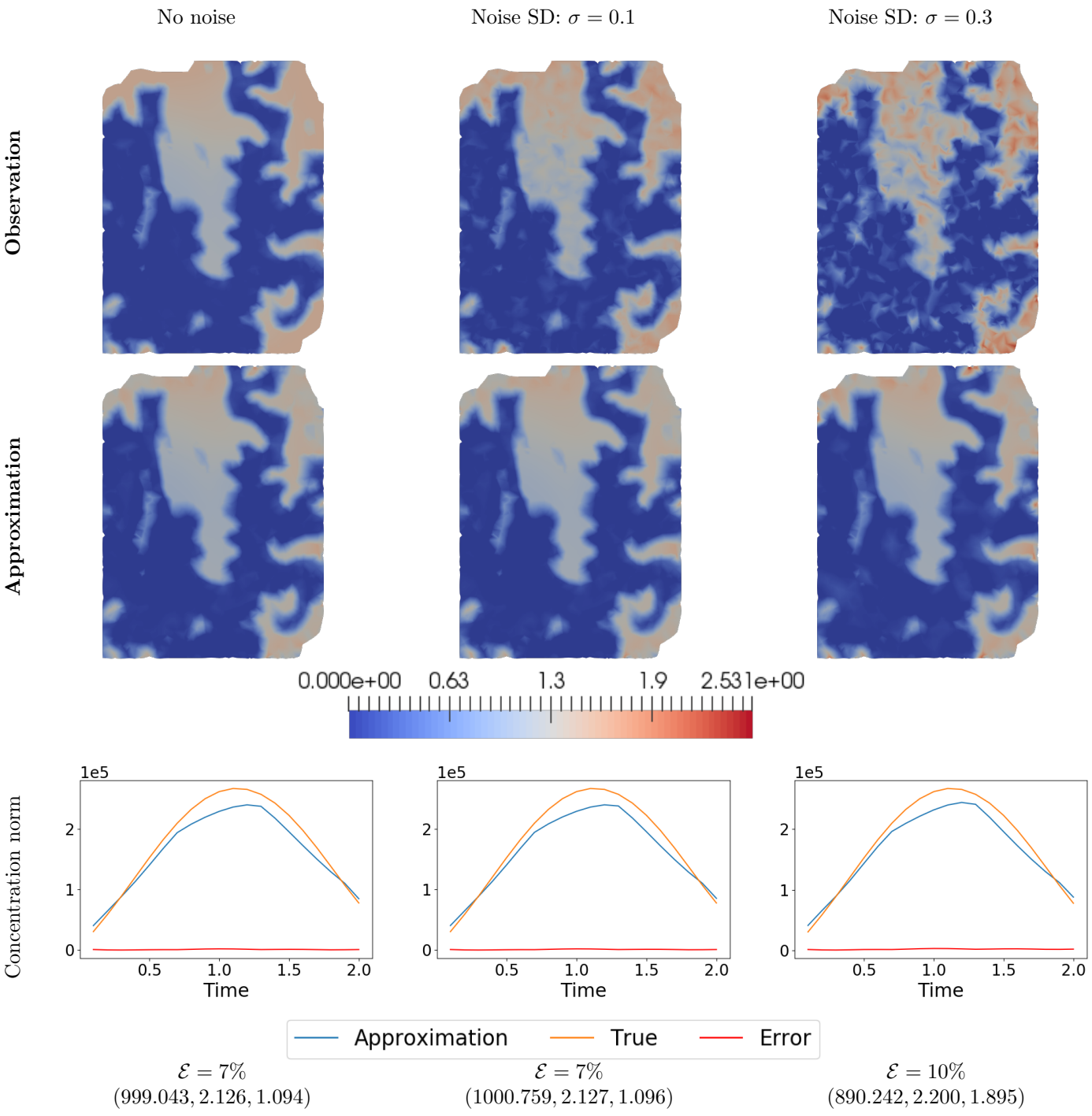


Figure 5.5: Results with different levels of pointwise Gaussian noise added to the observations. The standard deviation (SD) is written at the top. The top two rows feature cross-section snapshots at $t = 1.0$.

Chapter 6

Outlook: Mixed-models with Tensorflow

In this chapter we apply the generic AD framework implemented in chapter 3 to the open source software library TensorFlow [1]. TensorFlow is a machine learning library for python that implements reverse mode AD to compute gradients. For our purposes we will only consider TensorFlow as implementing neural networks, even though it can be used for other things. The chapter provides a brief outlook on the combination of neural networks and PDEs, which will be called mixed models.

6.1 Tensorflow and neural networks

Neural networks consists of an input layer and an output layer, together with zero or more hidden layers. Each layer consists of one or more nodes (or neurons), all of which take in some input and produces some output. This structure can be though of as a graph like illustrated in figure 6.1.

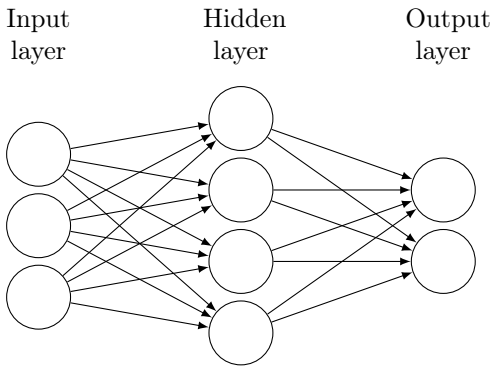


Figure 6.1: An illustration of a neural network with one hidden layer. Because each neuron takes input from every neuron in the previous layer, the neural network is called fully-connected.

Using figure 6.1 as reference, we illustrate the operations that could make up this neural network. The network takes three inputs, and all neurons in the hidden layer receive all these inputs multiplied by some weights. The weights can be thought of as applied along the edges. Thus, each neuron in the hidden layer results in some linear transformation with these weights $w_i \in \mathbb{R}^3$ on the input $x \in \mathbb{R}^3$.

$$a_i(x) = x^T w_i + b_i$$

The number $b_i \in \mathbb{R}$ is called the bias, where i denotes the index of the neuron in the hidden layer. In addition, a hidden layer neuron introduces a non-linear activation function on the output of the neuron. This activation function can for example be $h(a) = \tanh(a)$ or rectified linear unit (ReLU) $h(a) = \max(0, a)$. Thus the output from each neuron in the hidden layer can be summarised as $z_i = h(a_i(x))$. Reformulated, the hidden layer performs

$$z(x) = h(a(x)) = h(Wx + b)$$

where $W \in \mathbb{R}^{4\times3}$, $b \in \mathbb{R}^4$ and $h(a)$ applies the activation function element-wise. This is then used as input to the output layer where it is multiplied by another set of weights along the edges. Sometimes a non-linear function is also applied at the output layer, typically to restrict the output to a certain range of values. This is for example useful if the output should be probabilities.

As the input is usually a set of training data, x is thought of as an input matrix containing all the n examples $x \in \mathbb{R}^{n\times3}$. Then the model can be summarised as a mapping from the inputs $x \in \mathbb{R}^{n\times3}$ to the outputs $y \in \mathbb{R}^{n\times2}$ by

$$\begin{aligned} a_1 &= xW_1 + b_1 \\ z_1 &= \tanh(a_1) \\ a_2 &= z_1W_2 + b_2 \\ y &= z_2 = a_2 \end{aligned}$$

where we did not use any non-linear function for the output layer. This can be implemented in Tensorflow by

```
1  import tensorflow as tf
2
3  x = tf.placeholder(...)
4
5  W1 = tf.Variable(...)
6  b1 = tf.Variable(...)
7  a1 = tf.matmul(x, W1) + b1
8  z1 = tf.tanh(a1)
9
10 W2 = tf.Variable(...)
11 b2 = tf.Variable(...)
12 z2 = tf.matmul(z1, W2) + b2
13 y = z2
```

where the arguments to `tf.Variable` specify some initial value and thereby the shape and data type.

The training of this neural network can be formulated as a minimization problem. Given some inputs and corresponding outputs, we seek to minimize

some difference between computed outputs and training outputs $d \in \mathbb{R}^{n\times 2}$:

$$\min_{W_1,W_2,b_1,b_2} L(y,d)$$

where $L(y,d)$ is a *loss function* that measures the error, this could for example be a function for the mean squared error. What loss function is appropriate depends on the problem.

6.2 Mixed model approaches

In theory there are two ways one can combine two AD frameworks. Either we embed TensorFlow in pyadjont, or we embed pyadjoint in TensorFlow. Both ways involve implementing a function that represents a model from the other software package. This function works as a black box, both in the forward problem and in the adjoint problem. The only glaring difficulty is that the two models must be able to communicate. The black box function must accept input and give output that are valid in the driving software.

Another difficulty of mixing two AD frameworks is that they might not implement AD in the same way. For example, in TensorFlow the user builds the computational graph by using symbols and not values. Thus, the model is not run before it is initialized in a TensorFlow session with some values. This differs from pyadjoint, where the tape is built while the forward problem is computed. Our main focus will be on embedding pyadjoint in TensorFlow. It should be mentioned that TensorFlow offers an *eager mode* for debugging purposes, in which operations are executed immediately.

6.3 Implementing a FEniCS model in TensorFlow

For registering a custom function in TensorFlow we employ `py_func(func, inp, Tout, name=None, grad=None)` taken from [4]. This function wraps the function `func` as a TensorFlow operation with a gradient operation `grad`, taking the tensors `inp` as input and returning data types `Tout`. The gradient operation also needs to be a TensorFlow operation, thus we use `tf.py_func` [40], which is essentially the same as `py_func` just without gradient information. The inputs and outputs of Tensorflow operations are numpy arrays. Thus, certain helper functions are implemented to convert numpy arrays to and from FEniCS data types. These helper functions are in turn annotated to properly connect outputs and controls.

6.4 Verification

To test our implementations, a Taylor remainders test is constructed. Because dolfin-adjoint is now embedded in TensorFlow, the Taylor test must be implemented in Tensorflow. For this verification we employ a neural network with one hidden layer with 20 neurons. After the output layer of the neural network, a PDE is placed. Thus the output layer of the neural network produces some input for the FEniCS model. The PDE is the Poisson problem over a unit interval mesh, with the source term being the output of the neural network

ϵ	Residual	Order	Gradient Residual	Gradient order
$1 \cdot 10^{-2}$	$1.6405 \cdot 10^{-1}$		$8.2500 \cdot 10^{-3}$	
$5 \cdot 10^{-3}$	$8.4086 \cdot 10^{-2}$	1.0	$2.0625 \cdot 10^{-3}$	2.0
$2.5 \cdot 10^{-3}$	$4.2559 \cdot 10^{-2}$	1.0	$5.1571 \cdot 10^{-4}$	2.0
$1.25 \cdot 10^{-3}$	$2.1408 \cdot 10^{-2}$	1.0	$1.2893 \cdot 10^{-4}$	2.0

Table 6.1: The Taylor remainders without gradient information and with gradient information for a mixed model problem. The computed convergence orders are as expected.

and with prescribed 0 on the boundary. The implementation of the FEniCS model is shown below.

```
0  def fenics_model(x):
1      y = x.copy()
2      for i in range(len(x)):
3          x_ = x[i]
4          U = Function(V)
5
6          # Convert input to FEniCS function
7          tensorflow_to_fenics(f, x_)
8
9          a = inner(grad(u), grad(v))*dx
10         L = f*v*dx
11         bc = DirichletBC(V, 0, "on_boundary")
12         solve(a == L, U, bc)
13         y[i] = fenics_to_tensorflow(U)
14     return y
```

The loop over the input is because typically all training examples are given as input. The mean squared error is chosen as the loss function

$$L(y, d) = \frac{1}{k} \sum_{i=1}^{k} (y_i - d_i)^2$$

where y_i are the coefficients of the computed solution of the Poisson PDE with the neural network source term, k is the amount of coefficients in the Poisson solution, and d_i is the training outputs for each $i = 1, 2, ..., k$. For this d_i was set to 1 for each i. Using random perturbation direction the Taylor remainders are shown in table 6.1. The computed convergence rates are as expected, which indicates that the mixed models have been implemented correctly.

6.5 Summary & Discussion

Neural networks have proven to be very efficient at image recognition [28] and speech recognition [36], and have become widely popular in recent years. However, little research has been done on combining physical models through PDEs with neural networks. As stated by Gary Marcus, a professor of psychology at New York University and founder of the machine learning start-up Geometric Intelligence, "[...] researchers in deep learning appear to have a very strong

bias against including prior knowledge even when (as in the case of physics) that prior knowledge is well known." [35]. This gives rise to an interesting research topic. However, because of the lack of research, it is difficult to say what benefit these mixed models could have.

We presented a proof of concept for combining PDEs solved with FEniCS and neural networks defined in Tensorflow. The robustness of the implementation with respect to more complex models has yet to be determined. There is a lot of work to be done to get a framework that can seamlessly combine PDEs in FEniCS and Tensorflow. However, with the implementation of a flexible and easy to use interface, it could spark research interest on the topic of mixed models.

Chapter 7

Summary & future work

In this thesis we have implemented a generic AD Python framework providing a common interface for packages it is applied to, providing features such as optimization and verification. This framework was then applied to the Python interface of the FEM framework FEniCS [29]. This resulted in an AD package that can automatically derive and solve the adjoint equations. Examples and tests were shown, verifying that the AD package is correctly implemented, and illustrating that minimal changes needed to original code. Additionally, benchmarks of the implementation suggested that the computational costs of automatically deriving and solving the adjoint equations are near optimal, however with limited data on the scalability to even larger problems. The AD package was also implemented with support for strong Dirichlet boundary condition controls. This was implemented through the computation of the adjoint solution at the boundary. Lastly, we briefly touched upon the combination of PDEs and neural networks, illustrating that the AD package can be used together with TensorFlow. This opens up room for future development and experiments with these mixed-models.

7.1 Future work

For future work, the generic AD framework pyadjoint can be packed with additional features such as checkpointing [16]. The checkpointing would help remedy memory limitations for large-scale problems.

The primary target for pyadjoint application was FEniCS, thus the pyadjoint implementation is heavily influenced by the needs that arose when applying pyadjoint to FEniCS. It could be interesting to test the pyadjoint design principles on a different Python package, such as numpy [39]. This could reveal flaws and limitations of pyadjoint as a generic AD framework. Additionally, having support for numpy in pyadjoint could also help remedy the limitations of large low level computational graphs by the use of vectorisation. Providing AD to numpy has been previously done in a purely Python implementation with autograd [14], and it could be informative to see how pyadjoint compares.

Only some of the main interface of DOLFIN [30] has been overloaded in dolfin-adjoint. At the time of writing, dolfin-adjoint does for example not annotate assignment of linear combinations to functions. Another example is that some specialised solver classes such as `KrylovSolver` has not been

overloaded. Work must be done to have dolfin-adjoint work out of the box for almost any program that is restricted to the standard DOLFIN Python interface.

Lastly, more work can be done on combining dolfin-adjoint and machine learning. The outlook in 6 provides a base for experimentation. In particular, there is a need for comparing convergence rate to an optimum of problems with and without physical models. Additionally there is a need for attaining information on when mixed models can be useful, which can be gleaned from experimentation.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, and et al. Matthieu Devin. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie Rognes, and Garth Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[3] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37, March 2014.

[4] Heikki Arponen. Tensorflow custom function with gradient. `https://gist.github.com/harpone/3453185b41d8d985356cbe5e57d67342`, Accessed 09/05/2018.

[5] Yuri Bazilevs and Thomas JR Hughes. Weak imposition of dirichlet boundary conditions in fluid mechanics. *Computers & Fluids*, 36(1):12–26, 2007.

[6] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[7] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[8] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011.

[9] Lawrence C Evans. *Partial Differential Equations*. American Math Society, 2nd edition, 2010.

[10] Patrick E Farrell, David A Ham, Simon W Funke, and Marie E Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, 2013.

[11] Richard S Finkel, Claudia A Chiriboga, Jiri Vajsar, John W Day, Jacqueline Montes, Darryl C De Vivo, Mason Yamashita, Frank Rigo, Gene Hung, and Eugene et al. Schneider. Treatment of infantile-onset spinal muscular atrophy with nusinersen: a phase 2, open-label, dose-escalation study. *The Lancet*, 388(10063):3017–3026, 2016.

[12] Simon W Funke and Patrick E Farrell. A framework for automated pde-constrained optimisation. *arXiv preprint arXiv:1302.3894*, 2013.

[13] Michael B Giles and Niles A Pierce. An introduction to the adjoint approach to design. *Flow, turbulence and combustion*, 65(3-4):393–415, 2000.

[14] Github. Autograd. `https://github.com/HIPS/autograd`, Accessed 09/05/2018.

[15] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software*, 1(1):35–54, 1992.

[16] Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[17] Per Thomas Haga, Giulia Pizzichelli, Mikael Mortensen, Miroslav Kuchta, Soroush Heidari Pahlavian, Edoardo Sinibaldi, Bryn A Martin, and Kent-Andre Mardal. A numerical investigation of intrathecal isobaric drug dispersion within the cervical subarachnoid space. *PloS one*, 12(3):e0173680, 2017.

[18] Patrick Heimbach, Chris Hill, and Ralf Giering. An efficient exact adjoint of the parallel mit general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.

[19] Michael Hinze, Rene Pinnau, Michael Ulbrich, and Stefan Ulbrich. *Optimization with PDE constraints, vol. 23 of Mathematical Modelling: Theory and Applications*. Springer, New York, 2009.

[20] Karl Erik Holter, Benjamin Kehlet, Anna Devor, Terrence J Sejnowski, Anders M Dale, Stig W Omholt, Ole Petter Ottersen, Erlend Arnulf Nagelhus, Kent-André Mardal, and Klas H Pettersen. Interstitial solute transport in 3d reconstructed neuropil occurs by diffusion rather than bulk flow. *Proceedings of the National Academy of Sciences*, page 201706942, 2017.

[21] Jeffrey J Iliff, Minghuan Wang, Yonghong Liao, Benjamin A Plogg, Weiguo Peng, Georg A Gundersen, Helene Benveniste, G Edward Vates, Rashid Deane, and Steven A et al. Goldman. A paravascular pathway facilitates csf flow through the brain parenchyma and the clearance of interstitial solutes, including amyloid β. *Science translational medicine*, 4(147):147ra111–147ra111, 2012.

[22] Jocelyn Iott, Raphael T Haftka, and Howard M Adelman. Selecting step sizes in sensitivity analysis by finite differences. 1985.

[23] Antony Jameson. Aerodynamic design via control theory. *Journal of scientific computing*, 3(3):233–260, 1988.

[24] Byung-Ju Jin, Alex J Smith, and Alan S Verkman. Spatial model of convective solute transport in brain extracellular space does not support a "glymphatic" mechanism. *The Journal of general physiology*, 148(6):489–501, 2016.

[25] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>].

[26] R. N. King, K. Dykes, P. Graf, and P. E. Hamlington. Optimization of wind plant layouts using an adjoint approach. *Wind Energy Science*, 2(1):115–131, 2017.

[27] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, September 2006.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[29] Anders Logg, Kent-Andre Mardal, and Garth N. Wells et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[30] Anders Logg and Garth N. Wells. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):20:1–20:28, April 2010.

[31] Martin Losch and Patrick Heimbach. Adjoint sensitivity of an ocean general circulation model to bottom topography. *Journal of Physical Oceanography*, 37(2):377–393, 2007.

[32] Antoine Louveau, Sandro Da Mesquita, and Jonathan Kipnis. Lymphatics in neurological disorders: a neuro-lympho-vascular component of multiple sclerosis and alzheimer's disease? *Neuron*, 91(5):957–973, 2016.

[33] Zhoujie Lyu, Gaetan K Kenway, Cody Paige, and Joaquim Martins. Automatic differentiation adjoint of the reynolds-averaged navier-stokes equations with a turbulence model. In *21st AIAA Computational Fluid Dynamics Conference*, page 2581, 2013.

[34] Zhoujie Lyu, Gaetan KW Kenway, and Joaquim RRA Martins. Aerodynamic shape optimization investigations of the common research model wing benchmark. *AIAA Journal*, 53(4):968–985, 2014.

[35] Gary Marcus. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*, 2018.

[36] Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černockỳ. Strategies for training large scale neural network language models. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 196–201. IEEE, 2011.

[37] Stephen G Nash. Newton-type minimization via the lanczos method. *SIAM Journal on Numerical Analysis*, 21(4):770–788, 1984.

[38] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*, volume 24. Siam, 2012.

[39] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[40] TensorFlow API r1.8. tf.py_func. `https://www.tensorflow.org/api_docs/python/tf/py_func`, Accessed 09/05/2018.

[41] Olivier Talagrand. Assimilation of observations, an introduction. *Journal of the Meteorological Society of Japan. Ser. II*, 75(1B):191–209, 1997.

[42] Markus Towara and Uwe Naumann. A discrete adjoint model for openfoam. *Procedia Computer Science*, 18:429–438, 2013.

[43] Daniel J Wolak and Robert G Thorne. Diffusion of macromolecules in the brain: implications for drug delivery. *Molecular pharmaceutics*, 10(5):1492–1504, 2013.

[44] David W Zingg, Marian Nemec, and Thomas H Pulliam. A comparative evaluation of genetic and gradient-based algorithms applied to aerodynamic optimization. *European Journal of Computational Mechanics/Revue Européenne de Mécanique Numérique*, 17(1-2):103–126, 2008.