A Review of Sensitivity Methods for Differential Equations

Facundo Sapienza*[1], Jordi Bolibar[2], Frank Schäfer[3], Patrick Heimbach[6], Giles Hooker[4], Fernando Pérez[1], and Per-Olof Persson[5]

[1]*Department of Statistics, University of California, Berkeley (USA)*
[2]*TU Delft, Department of Geosciences and Civil Engineering, Delft (Netherlands)*
[3]*CSAIL, Massachusetts Institute of Technology, Cambridge (USA)*
[4]*Department of Statistics and Data Science, University of Pennsylvania (USA)*
[5]*Department of Mathematics, University of California, Berkeley (USA)*
[6]*Department of Earth and Planetary Sciences, Jackson School of Geosciences, University of Texas, Austin (USA)*

December 2, 2023

*Corresponding author: fsapienza@berkeley.edu

Abstract

The differentiable programming paradigm has become a central component of modern machine learning techniques. A long tradition of this paradigm exists in the context of scientific computing, in particular in partial differential equation-constrained, gradient-based optimization. The recognition of the strong conceptual synergies between inverse methods and machine learning offers the opportunity to lay out a coherent framework applicable to both fields. For models described by differential equations, the calculation of sensitivities and gradients requires careful algebraic and numeric manipulations of the underlying dynamical system. Here, we provide a comprehensive review of existing techniques to compute gradients of numerical solutions of differential equation systems. We first discuss the importance of gradients of solutions of ODEs in a variety of scientific domains, covering computational fluid dynamics, electromagnetism, geosciences, meteorology, oceanograpgy, climate science, flux inversion, glaciology, solid earth geophysics, biology and ecology, and quantum physics. Second, we lay out the mathematical foundations of the different approaches. Finally, we discuss the computational consideration and solutions that exist in modern scientific software.

Long context for an abstract. We could synthesize this, going more straight to the point of the paper. I would also emphasize on the importance of solution sensitivity in science.

To the community, by the community. *This manuscript was conceived with the goal of shortening the gap between developers and practitioners of differentiable programming applied to modern scientific machine learning. With the advent of new tools and new software, it is important to create pedagogical content that allows the broader community to understand and integrate these methods into their workflows. We hope this encourages new people to be an active part of the ecosystem, by using and developing open-source tools. This work was done under the premise* ***open-science from scratch****, meaning all the contents of this work, both code and text, have been in the open from the beginning and that any interested person can contribute to the project. You*

I feel like the motivations are as impor-

can contribute directly to the GitHub repository `github.com/ODINN-SciML/DiffEqSensitivity-Review`.

Contents

Plain language summary

Differential equations are mathematical tools that explicitly describe the processes and dynamics within various systems, based on prior knowledge. They are fundamental in many scientific disciplines for modeling phenomena such as physical processes, population dynamics, social interactions, and chemical reactions.

here, it would be nice to explain how they differ to data-drive models. Consider:

By contrast, data-driven models do not necessarily require a detailed understanding of the underlying processes, and learn patterns and relationships directly from data. Data-driven models are particularly useful in scenarios where the underlying processes are poorly understood or too complex to be captured by traditional mathematical models. The combination of mechanistic models with data-driven models is becoming increasingly common in many scientific domains. In order to achieve this, these models need to leverage both domain knowledge and data, in order to have an accurate representation of the underlying dynamics. Being able to determine which model parameters are most influential and further compute derivatives of such a model is key to correctly assimilating and learning from data, but a myriad of sensitivity methods exist to do so. We provide an overview of the different sensitivity methods that exist, providing (i) guidelines on the best use cases for different scientific domain problems, (ii) detailed mathematical analyses of their characteristics, and (iii) computational implementations on how to solve them efficiently.

1 Introduction

I would argue that the introduction should actually be the scientific motivation

it would be useful to state here how this paper differs from[96]

. Evaluating how the value of a function changes with respect to its arguments and parameters plays a central role in optimization, sensitivity analysis, Bayesian inference, inverse methods, and uncertainty quantification, among many.[72] Modern machine learning applications require the use of gradients to efficiently exploit the high-dimensional space of parameters to be inferred or learned (e.g., weights of a neural network). The same is true for partial differential equation (PDE) constrained optimization, which infers uncertain (or unknown) parameters from (generally sparse) data.[24] When optimizing an objective function (loss function in machine learning or cost function in inverse modeling), gradient-based methods (for example, gradient descent and its many variants[75]) are more efficient at finding a minimum and converge faster to them than gradient-free methods. Furthermore, the *curse of dimensionality* renders gradient-free optimization and sampling methods computationally intractable for most large-scale problems.[64] When numerically computing the posterior of a probabilistic model, gradient-based sampling strategies converge faster to the posterior distribution than gradient-free methods. Hessians further help to improve the convergence rates of these algorithms and enable uncertainty quantification around parameter values.[12] *A gradient serves as a compass in modern data science: it tells us in which direction in the vast, open ocean of parameters we should move towards in order to increase our chances of success.*

Dynamical systems governed by differential equations are not an exception to the rule. Differential equations play a central role in describing the behaviour of systems in natural and social sciences. Some authors have recently suggested differentiable programming as the bridge between modern machine learning methods and traditional scientific models.[22,71,82] Being able to compute gradients or sensitivities of dynamical systems opens the door to more complex models. This is very appealing in geophysical models, where there is a broad literature on physical models and a long tradition in numerical methods. The first goal of this work is to introduce some of the applications of this emerging technology and to motivate its incorporation for the modelling of complex systems in the natural and social sciences.

which rule? Here we could say that the solution of differential equations can be seen as functions which map parameter and initial conditions to state variables, similarly to machine learning models.

Question 1. *What are the scientific applications of differentiable programming for complex dynamical systems?*

- this paragraph that introduces sensitivity analysis and its challenges seems disconnected from the question, which relates to differentiable programming. the question seems more relevant to the second paragraph, which introduces AD
- I would first have a question on the usefulness of calculating parameter sensivitity, and then bring sensitivity analysis and its associated challenges

Sensitivity analysis corresponds to any method aiming to calculate how much the output of a function or program changes when we vary one of the function (or model) parameters. This task is performed in different ways by different communities when working with dynamical systems. In statistics, the sensitivity equations enable the computation of gradients of the likelihood of the model with respect to the parameters of the dynamical system, which can be later used for inference.[70] In numerical analysis, sensitivities quantify how the solution of a differential equation fluctuates with respect to certain parameters. This is particularly useful in optimal control theory,[27] where the goal is to find the optimal value of some control (e.g. the shape of a wing) that minimizes a given loss function. In recent years, there has been an increasing interest in designing machine learning workflows that include constraints in the form of differential equations. Examples of this include methods that numerically solve differential equations, such as physics-informed neural networks,[69] as well as methods that augment and learn parts of the differential equation, such as universal differential equations,[18,68] which also includes the case of neural ODEs[13] and neural SDEs.[51] Furthermore, numerical solvers are used as forward models in the case of neural ordinary differential equations.[13]

However, when working with differential equations, the computation of gradients is not an easy task, both regarding the mathematical framework and software implementation involved. Except for a small set of particular cases, most differential equations require numerical methods to calculate their solution and cannot be differentiated analytically. This means that solutions cannot be directly differentiated and require special treatment to compute first or second-order derivatives. Furthermore, numerical solutions introduce approximation errors. These errors can be propagated and amplified during the computation of the gradient. Alternatively, there is a broad literature on numerical methods for solving differential equations. Although each method provides different guarantees and advantages depending

on the use case, this means that the tools developed to compute gradients when using a solver need to be universal enough in order to be applied to all or at least to a large set of them. The second goal of this article is to review different methods that exist to achieve this goal.

> **Question 2.** *How to compute the gradient of a function that depends on the numerical solution of a differential equation?*

The broader set of tools known as Automatic (or Algorithmic) Differentiation (AD) aims at computing derivatives by sequentially applying the chain rule to the sequence of unit operations that constitute a computer program.[32,61] The premise is simple: every computer program, including a numerical solver, is ultimately an algorithm described by a chain of elementary algebraic operations (addition, multiplication) that are easy to differentiate and their combination is easy to differentiate by using the chain rule.[25] Although many modern differentiation tools use AD to some extent, there is also a family of methods that compute the gradient by relying on an auxiliary set of differential equations. We are going to refer to this family of methods as *continuous*, and we will dedicate them a special treatment in future sections to distinguish them from the discrete algorithms that resemble more to pure AD. We emphasize that AD is merely a *tool* for generating derivatives of computer code. Alternatively, such code can be (and often has been) generated "by hand", e.g., to avoid restrictions incurred by the available AD tool. Down-sides of hand-written derivative code are, (i) it is error-prone, (ii) it is difficult to keep pace with the development of the parent code, and (iii) it enables continuous development of the models.

The differences between methods arise both from their mathematical formulation and their computational implementation. The first provides different guarantees on the method returning the actual gradient or a good approximation thereof. The second involves how theory is translated to software, and what are the data structures and algorithms used to implement it. Different methods have different computational complexities depending on the number of parameters and differential equations, and these complexities are also balanced between total execution time and required memory. The third goal of this work, then, is to illustrate the different strengths and weaknesses of these methods, and how to use them in modern scientific software.

> **Question 3.** *What are the advantages and disadvantages of different differentiation methods and how can I incorporate them in my research?*

No connection between Julia and the above question.

Despite the fact that these methods can be (in principle) implemented in different programming languages, here we decided to use the Julia programming language for the different examples. Julia is a recent but mature programming language that has already a large tradition in implementing packages aiming to advance differentiable programming.[8,9] Nevertheless, in reviewing existing work, we also point to applications developed in other programming languages.

Without aiming at making an extensive and specialized review of the field, we believe this study will be useful to other researchers working on problems that combine optimization and sensitivity analysis with differential equations. Differentiable programming is opening

new ways of doing research across sciences, and we need close collaboration between domain scientists, methodological scientists, computational scientists, and computer scientists in order to develop successful, scalable, practical, and efficient frameworks for real-worlds applications.[Frank2022] As we make progress in the use of these tools, new methodological questions start to emerge. How do these methods compare? How can they be improved?

It would be nice to have a paragraph which announces the structure of the paper. Overall, I also feel like the structure in the form of questions could be whether improved, or paragraph could simply be concatenated without questions.

2 Inverse modelling in science

2.1 Inverse modelling

here I would focus on the philosophy of mechanistic vs data-driven models

Scientific models from many domains have often been based on mechanistic (or process-based) models, represented as differential equations, involving the use of numerical methods to solve them. .

- Albert Einstein once said DEs are the most important conceptual advance ever made in human history
- Differential Equations start in 17th century with with Isaac Newton

I would include this part in the introduction

From predictions of population growth (Malthus) to permitting the prediction of black holes (Einstein field equations), differential equations have provided huge scientific contributions since Newton. The parameters and processes within process-based models have traditionally been determined independently of the model, with empirical data for model validation and comparison.[hartig2012] Independent estimation of parameters and processes rapidly becomes impossible as the number of state variables modelled increases, especially when considering highly non-linear processes. Inverse modelling, which consists in using observation data to recover the parameters of a model that can best explain the data, allows the bridging of this gap.[74,93] The process knowledge embedded in the structure of mechanistic models renders them more robust for predicting dynamics under different conditions. Nonetheless, in the 21st century, with the unstoppable wave of data flooding all scientific domains, progress with such traditional methods has become more complex.

In parallel, the field of statistics experienced a boom following the massive growth of data, signaling the era of data science and machine learning.[16] With the advent of machine learning methods, it is possible to learn and capture extremely complex nonlinear patterns and information hidden in huge datasets. Machine learning models can be seen as the opposite of mechanistic models: they are flexible, data-driven and they do not necessarily respect domain-specific constraints. .

a big aspect of modern machine (deep) learn-

At first sight, these two modelling philosophies can be seen as antagonistic, and this is more or less the way they have evolved in the last decades.[95] On the one hand, domain scientists have often been sceptical of adopting machine learning methods, judging them as opaque

black boxes, unreliable, and not respecting domain-established knowledge.[15] Predictions with correlative models assume that patterns contained in data can be extrapolated.**dormann2007** However, they may fail to disentangle the respective impact of the numerous ecological processes at play and may fail to predict dramatic shifts in dynamics.**Barnosky2012** On the other hand, the field of machine learning has mainly been developed around data-driven applications, without including any *a priori* physical knowledge. However, there has been an increasing interest in making mechanistic models more flexible, as well as introducing domain-specific or physical constraints and interpretability in machine learning models.**Schneider2017**, **rasp2018**, **Yazdani2020**, **Abarbanel2018**, **Carrassi2018**, **Bocquet2019**, **Gabor2015**, **Gharamti2017**, **Curtsdotter**

2.2 Challenges

here I would actually mention sensitivity analysis as a way to bridge the two worlds. Then I would introduce differentiable programming as a game changer to performe sensitivity analysis.

A key way to achieve this is through differentiable programming, i.e. being able to compute derivatives of any computer program describing a scientific model. During the last decades, the backpropagation algorithm has enabled the fast-growing of deep learning by efficiently computing gradients of large and complex neural networks with many parameters.[31] Nowadays, the differentiation of hybrid models comprising data-driven models (e.g. neural networks, gaussian processes) with differential equations poses complex technical problems, which are only starting to be explored in recent years.[54] Being able to accurately estimate model parameters, ranging from a few ones in classic inversion problems to millions of them in large neural networks, opens many new possibilities. Differentiable programming has the potential to revolutionize the way we approach and design scientific models and even the way we discover governing laws from data.

2.3 Domain-specific applications

Differential equations can be used to describe a large variety of dynamical systems, while data-driven regression models (e.g., neural networks, Gaussian processes, reduced-order models, basis expansions) have been demonstrated to act as universal approximators, learning any possible function if enough data is available.[29] This combined flexibility can be exploited by many different domain-specific problems to tailor modelling needs to both dynamics and data characteristics.

2.3.1 Computational Fluid Dynamics

2.3.2 Electromagnetism

2.3.3 Geosciences

Many geoscientific phenomena are governed by global and local conservation laws (conservation of mass, momentum, energy, tracers) along with a set of empirical constitutive laws and subgrid-scale parametrization schemes. Together, they enable efficient description of the

system's spatio-temporal evolution in terms of a set of partial differential equations (PDEs). Example are geophysical fluid dynamics,[88] describing geophysical properties of many Earth systems, such as the atmosphere, oceans, and glaciers. In such models, calibrating model parameters is extremely challenging, due to datasets being sparse in both space and time, heterogeneous, and noisy. Moreover, many existing mechanistic models can only partially describe observations, with many detailed physical processes being ignored or poorly parameterized. The use of differentiable programming, combining PDEs and data-driven models (i.e. Universal Differential Equations) may add flexibility to mechanistic models in order to incorporate new governing laws from data (from either measurement or simulations).[68]

Arguably, the notion of differentiable programming has a long tradition in the geosciences in the context of solving large-scale geophysical inverse problems. The overarching goal of such problems is to find a set of optimal model parameters that minimize a (usually weighted least-squares) objective or cost function quantifying the misfit between observations and the simulated state, subject to the constraint that the model equations be fulfilled. The constrained optimization problem is transformed into an unconstrained problem by way of *Lagrange multiplier method*, also referred to as the *adjoint method*. The corresponding *adjoint model* computes the gradient of the objective function with respect to all inputs. Gradient-based nonlinear optimization then enables us to "invert" for optimal values of the unknown or uncertain inputs. Depending on the nature of the inputs, we may distinguish the following cases:

- *Initial conditions:* Inverting for uncertain initial conditions, which, when integrated using the model, lead to an optimal match of the observations; variants thereof are used for optimal forecasting (see below);
- *Boundary conditions:* Inverting for uncertain surface, bottom, or lateral boundaries (e.g., open boundaries of a limited domain), which, when used in the model, produce an optimal match of the observations; variants thereof are used in tracer or boundary (air-sea) flux inversion problems, e.g., related to the global carbon cycle;
- *Model parameters:* Inverting for uncertain model parameters amounts to an optimal model calibration problem. As a "learning of optimal parameters from data" problem, it is the closest to machine learning applications.

In addition to the use of gradients or derivative information for optimization, inversion, estimation, or "learning", gradients have also proven powerful tools for

- computing *comprehensive sensitivities* of quantities of interest,
- computing *optimal perturbations* (in initial or boundary conditions) that lead to maximum amplification of specific norms of interest,
- characterizing and quantifying uncertainties by way of second derivative (Hessian) information.

Within the framework of gradient-based inversion, all of these cases rely on the availability of an adjoint model of the (in general nonlinear) geophysical parent model to efficiently compute the gradient of the objective function with respect to a usually very high-dimensional

(typically $O(10^3) - O(10^8)$) space of inputs. In the following, we sketch how differentiable programming - from the perspective of adjoint modeling - has been used in different disciplines of geosciences, and how new concepts are emerging of combining inverse modeling and machine learning approaches where differentiable programming provides a key computational enabling framework. (Note that some authors have used the notion of "scientific machine learning" to capture some aspects of the latter approach [REFS]).

Meteorology ...

Oceanography ...

Climate science ...

Flux inversion ...

Glaciology Glaciers act as slow fluids, flowing down-slope through the effects of gravity, and the understanding of their rheological properties (e.g. ice viscosity affecting internal deformation or sliding at the glacier-bedrock interface) is key to assessing their contribution to water resources and sea-level rise.[17] These rheological processes and their dependency on key large-scale environmental variables, such as the local climate or topography, are still not well understood. Recent studies had showed how neural networks integrated with differential equations can be used to solve ice flow equations[42,43] and invert rheological properties at the same time.[10,91]

Solid Earth geophysics ...

2.3.4 Biology and Ecology

Here I am happy to provide a similar paragraph as the one above (Geosciences), although my expertise is mostly restrained on ecological modelling.

2.3.5 Quantum Physics

3 Statistical foundations of inverse modelling

Here, I would move the subsection "Preliminaries" from the next section.

3.1 Models and data

System of ordinary differential equations (ODEs) can generally be described as

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), p) \\ x(0) &= x_0 \\ y(t) &= h(x(t)) + \epsilon(t) \end{aligned} \tag{1}$$

where $x(t) \in \mathbb{R}^m$ is a vector of state variables that might represent XXX, $y(t) \in \mathbb{R}^d$ is a vector of observables that contains a subset or aggregates of the state variables, and $p \in \mathbb{R}^q$ is the model parameter vector. h is a function that maps the state variables to the observables, which may be contaminated with noise ϵ, here of Gaussian type, with zero mean and variance–covariance matrix Σ_y. Denoting by $\theta = (x_0, p)$ the vector containing the ICs and the parameters, the model may be viewed as a map $\mathcal{M}$ parametrized by time t that takes the parameters θ to the state variables x

$$\begin{aligned} \mathcal{M}(t, \theta) &= x(t) \\ &= \int_0^t f(s, x(s), p) ds + x_0 \end{aligned} \tag{2}$$

3.2 Maximum likelihood estimation and loss function

Taking expectations over the noise realizations yields $\mathbb{E}[y(t)] = h(\mathcal{M}(t, \theta))$, and it follows that the conditional likelihood of each observation $y_k \equiv y(t_k)$, given the parameters θ and the model $\mathcal{M}$ denoted by $p(y_k|\theta, \mathcal{M})$, follows the distribution of the residuals $\epsilon_k \equiv \epsilon(t_k) = y(t_k) - h\left(\mathcal{M}(t_k, \theta)\right)$, which corresponds to the multivariate normal distribution $\mathcal{N}_{0,\Sigma_y}$. Following a Bayesian approach, the calibration of the model can be performed on the basis of the parameter and model posterior probability $p(\theta, \mathcal{M}|\mathbf{y}_{1:K})$, i.e. the conditional probability density of the parameter values θ and the model $\mathcal{M}$ given the data, given by

$$p(\theta, \mathcal{M}|\mathbf{y}_{1:K}) \propto p(\mathbf{y}_{1:K}|\theta, \mathcal{M}) p(\theta, \mathcal{M}) \tag{3}$$

where $\mathbf{y}_{1:K} = (y_1, \ldots, y_K)$, $p(\mathbf{y}_{1:K}|\theta, \mathcal{M})$ is the product of the conditional likelihood of each observation y_k

$$\begin{aligned} p(\mathbf{y}_{1:K}|\theta, \mathcal{M}) &= \prod_{i=1}^{K} p(y_i|\theta, \mathcal{M}) \\ &= \prod_{k=1}^{K} \frac{1}{\sqrt{(2\pi)^d |\Sigma_y|}} \exp\left(-\frac{1}{2} \epsilon_k^T \Sigma_y^{-1} \epsilon_k\right) \end{aligned} \tag{4}$$

and $p(\theta, \mathcal{M})$ is the prior distribution of the model and its associated parameter values. The model $\mathcal{M}$ is included in the probabilistic quantities in order to accommodate multiple candidate models.

A variational method to obtain a Bayesian estimate of θ involves maximizing $p(\theta, \mathcal{M}|\mathbf{y}_{1:k})$ to obtain the maximum a posteriori (MAP) estimator,**Bocquet2019** which is equivalent to a maximum likelihood approach under a uniform prior distribution of the parameters, i.e. when no prior information on the parameter values is used.**Schartau2017** Observing that maximizing $p(\theta, \mathcal{M}|\mathbf{y}_{1:K})$ is equivalent to minimizing $-\log p(\theta|\mathbf{y}_{1:K}, \mathcal{M})$ and assuming a normal prior distribution of the parameters $\mathcal{N}_{p_b,\Sigma_p}$, one can obtain the MAP $\hat{\theta}$

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} L_{\mathcal{M}}(\theta) \tag{5}$$

where $L_{\mathcal{M}}$ is referred to as the loss function and defined as

$$L_{\mathcal{M}}(\theta) = \frac{1}{2}\left[\sum_{k=1}^{K-1} \|y_k - h\left(\mathcal{M}(t_k,\theta)\right)\|^2_{\Sigma_y} + \|p - p_b\|^2_{\Sigma_p}\right] \tag{6}$$

Schneider2017, **Raue2009** and where we use the notation $\|y\|^2_{\Sigma} = y\Sigma^{-1}y^{\mathbf{T}}$. Eq. 6 is similar to a traditional least squares function commonly used in regression, where the second summand is the analogue of a regularization term for the weights and biases of e.g. a neural network.

Gradient-based optimizers can then be used to efficiently obtain $\hat{\theta}$ in Eq. 5, iteratively updating the parameter vector θ_m given the gradient of the loss function, denoted by $\nabla_\theta L_{\mathcal{M}}$, to navigate the surface defined by $L_{\mathcal{M}}$ with the aim to find the global minimum where $\nabla_\theta L_{\mathcal{M}}(\hat{\theta}) = 0$. As an example, the plain vanilla gradient descent algorithm is given by

$$\theta_{m+1} = \theta_m - \gamma \nabla_\theta L_{\mathcal{M}}(\theta_m) \tag{7}$$

where γ is the learning rate. Other gradient-based algorithms, such as the ADAM optimizer used in the section below, employ more advanced updating strategies to avoid convergence to local minima but stay in the spirit of Eq. 7.

3.3 Likelihood profiles

3.4 Quantity of interest

3.5 Diagnosis of the solution

4 Methods

Depending on the number of parameters and the complexity of the differential equation we are trying to solve, there are different methods to compute gradients with different numerical and computational advantages. These methods can be roughly classified as:

- *Discrete* vs *continuous* methods
- *Forward* vs *backward* methods

The first difference regards the fact that the method for computing the gradient can be either based on the manipulation of atomic operations that are easy to differentiate using the chain rule several times (discrete), in opposition to the approach of approximating the gradient as the numerical solution of a new set of differential equations (continuous). Another way of conceptualizing this difference is by comparing them with the discretize-differentiate and differentiate-discretize approaches.[11,66,83,96] We can either discretize the original system of ODEs in order to numerically solve it and then define the set of adjoint equations on top of the numerical scheme; or instead define the adjoint equation directly using the differential equation and then discretize both in order to numerically approximate the solutions.[27]

The second distinction is related to the fact that some methods compute gradients by resolving a new sequential problem that may move in the same direction as the original

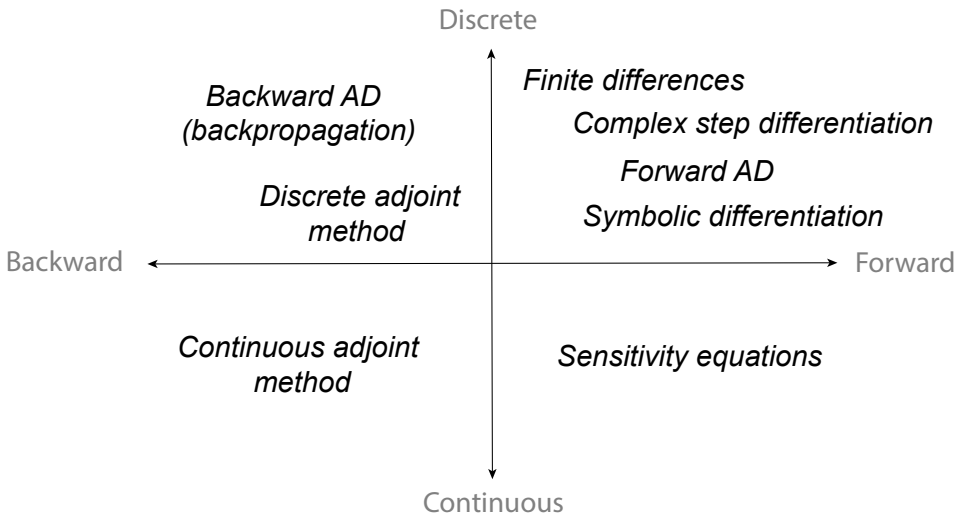

Figure 1: Schematic representation of the different methods available for differentiation involving differential equation solutions. These can be classified depending if they find the gradient by solving a new system of differential equations (*continuous*) or if instead they manipulate unit algebraic operations (*discrete*). Additionally, these methods can be categorized based on their alignment with the direction of the numerical solver. If they operate in the same direction as the solver, they are referred to as *forward methods*. Conversely, if they function in the opposite direction, they are known as *backward methods*.

numerical solver - i.e. moving forward in time - or, instead, they solve a new system that goes backwards in time. Figure 1 displays a classification of some methods under this two-fold classification. In the following section, we explore more in detail these methods.

It is important to note that if all the methods we explore in this section are mathematically correct, *that does not imply they are numerically stable*. These statements applied to methods based on pure automatic differentiation as well as adjoint methods. We explore this consideration in more detail in section 5.

seems a bit disconnected

4.1 Preliminaries

Consider a system of ordinary differential equations (ODEs) given by

$$\frac{du}{dt} = f(u, \theta, t), \tag{8}$$

where $u \in \mathbb{R}^n$ is the unknown solution; f is a function that depends on the state u, some parameter $\theta \in \mathbb{R}^p$, and an independent variable t which we will refer as time, but it can represent another quantity; and with initial condition $u(t_0) = u_0$. Here n denotes the total number of ODEs and p the size of a parameter embedded in the functional form of the differential equation. Although here we consider the case of ODEs, that is, when the derivatives are just with respect to the time variable t, the ideas presented here can be extended of the case of partial differential equations (for example, via the method of lines[1]) and algebraic differential equations (ADE). Except for a minority of functions $f(u, \theta, t)$, solutions of the Equation (8) need to be computed using a numerical solver.

We are interested in computing the gradient of a given function $L(u(\cdot,\theta))$ with respect to the parameter θ. Here we are using the letter L to emphasize that in many cases this will be a loss function, but without loss of generality this includes a broader class of functions.

- **Empirical loss functions**. This is usually a real-valued function that quantifies the distance between the model prediction and the data. Examples of loss functions include the squared error

$$L(u(\cdot,\theta)) = \frac{1}{2}\|u(t_1;\theta) - u^{\text{target}(t_1)}\|_2^2, \tag{9}$$

where $u^{\text{target}(t_1)}$ is the desired target observation at some later time t_1. More generally, we can evaluate the loss function at points of the time series for which we have observations,

$$L(u(\cdot,\theta)) = \frac{1}{2}\sum_{i=1}^{N}\|u(t_i;\theta) - u^{\text{target}(t_i)}\|_2^2. \tag{10}$$

We can also consider the continuous evaluated loss function of the form

$$L(u(\cdot,\theta)) = \int_{t_0}^{t_1} h(u(t;\theta),\theta))dt, \tag{11}$$

with h being a function that quantifies the contribution of the error term at every time $t \in [t_0, t_1]$. Defining a loss function where just the empirical error is penalized is known as trajectory matching. Other methods like gradient matching and generalized smoothing the loss depends on smooth approximations of the trajectory and their derivatives.

this is unclear

- **Likelihood profiles.** From a statistical perspective, it is common to assume that observations correspond to noisy observations of the underlying dynamical system, $y_i = u(t_i;\theta) + \varepsilon_i$, with ε_i errors or residual that are independent of each other and of the trajectory $u(\cdot;\theta)$.[70] If $p(y|t,\theta)$ is the probability distribution of y, maximum likelihood estimation consists in finding the parameter θ as

$$\theta^* = \underset{\theta}{\text{argmax}}\, \ell(y|\theta) = \prod_{i=1}^{n} p(y_i|\theta,t_i). \tag{12}$$

When $\varepsilon \sim N(0,\sigma_i^2)$ is Gaussian, the maximum likelihood principle is the same as minimizing $-\log \ell(y|\theta)$ which results in the mean squared error

$$\theta^* = \underset{\theta}{\text{argmin}}\, \{-\log \ell(y|\theta)\} = \underset{\theta}{\text{argmin}} \sum_{i=1}^{n} (y_i - u(t_i;\theta))^2\,. \tag{13}$$

There is a correspondance between this equation and the empirical loss function above, which would be nice to show. See the proposed section "Maximum likelihood estimation and loss function". This is also discussed in e.g. https://www.deeplearningbook.org/contents/prob.html

Provided with a prior distribution $p(\theta)$ for the parameter θ, we can further compute a posterior distribution for θ given the observations $y_1, y_2, \ldots, y_n$ following Bayes theorem

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)}. \tag{14}$$

In practice, the posterior is difficult to evaluate and needs to be approximated using Markov chain Monte Carlo sampling methods.[23] being able to further compute gradients of the likelihood allows to design more efficient sampling methods, such as Hamiltonian MCMC.[7]

- **Quantity of interest.** Another important example is when L returns the value of the solution at one or many points, which is useful when we want to know how the solution itself changes as we move the parameter values.
- **Diagnosis of the solution.** In many cases we are interested in optimizing the value of some variable that is a function of the solution of a differential equation. This is the case in design control theory, a popular approach in aerodynamics modelling where goals include maximizing the speed of an airplane or the lift of a wing given the solution of the flow equation for a given geometry profile.[39]

consider: model sensitivity to parameters

Notice that these cases also include sensitivities of L with respect to the initial condition of the system.

In the context of optimization, being able to compute sensitivities allows to perform gradient-based updates on the parameter θ by

$$\theta^{k+1} = \theta^k - \alpha_k \frac{dL}{d\theta^k}. \tag{15}$$

There exists different variants of gradient descent methods,[75] and in general they outperform gradient-free optimization schemes [REF]. Furthermore, gradient-free methods (also known as global optimization techniques) rely in heuristics[67] that are not guaranteed to find the solution.

Some gradient free methods are not necessarily global optimization techniques, e.g. evolutionary algorithms wilke2

Using the chain rule we can derive

$$\frac{dL}{d\theta} = \frac{dL}{du}\frac{\partial u}{\partial \theta}. \tag{16}$$

The first term on the right-hand side is usually easy to evaluate since it just involves the partial derivative of the scalar loss function with respect to the solution. For example, for the loss function in Equation (9) this is simply

$$\frac{dL}{du} = u - u^{\text{target}(t_1)}. \tag{17}$$

The second term on the right-hand side is more difficult to compute and it is usually referred to as the *sensitivity*,

$$s = \frac{\partial u}{\partial \theta} = \begin{bmatrix} \frac{\partial u_1}{\partial \theta_1} & \cdots & \frac{\partial u_1}{\partial \theta_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_n}{\partial \theta_1} & \cdots & \frac{\partial u_n}{\partial \theta_p} \end{bmatrix} \in \mathbb{R}^{n \times p}. \tag{18}$$

Notice here the distinction between the total derivative (indicated with the d) and partial derivative symbols (∂). When a function depends on more than one argument, we use the partial derivative symbol to emphasize this distinction (e.g., Equation (18)). On the other side, when this is not the case, we will use the total derivative symbol (e.g., Equation (17)). Also notice that the sensitivity s defined in Equation (18) is what is called a *Jacobian*, that is, a matrix of first derivatives for general vector-valued functions.

4.2 Finite differences

The simplest way of evaluating a derivative is by computing the difference between the evaluation of the function at a given point and a small perturbation of the function. In the case of a loss function, we can approximate

$$\frac{dL}{d\theta_i}(\theta) \approx \frac{L(\theta + \varepsilon e_i) - L(\theta)}{\varepsilon}, \tag{19}$$

with e_i the i-th canonical vector and ε the stepsize. Even better, it is easy to see that the centered difference scheme

$$\frac{dL}{d\theta_i}(\theta) \approx \frac{L(\theta + \varepsilon e_i) - L(\theta - \varepsilon e_i)}{2\varepsilon}, \tag{20}$$

leads also to a more precise estimation of the derivative. While Equation (19) gives to an error of magnitude $\mathcal{O}(\varepsilon)$, the centered differences schemes improves to $\mathcal{O}(\varepsilon^2)$.[2]

However, there are a series of problems associated with this approach. The first one is due to how this scales with the number of parameters p. Each directional derivative requires the evaluation of the loss function L twice. For the centered differences approach in Equation (20), this requires a total of $2p$ function evaluations, which at the same time demands solving the differential equation in forward mode each time for a new set of parameters.

A second problem is due to rounding errors. Every computer ultimately stores and manipulates numbers using floating points arithmetic.[28] Equations (19) and (20) involve the subtraction of two numbers that are very close to each other, which leads to large cancellation errors for small values of ε that are amplified by the division by ε. On the other hand, large values of the stepsize give inaccurate estimations of the gradient. Finding the optimal value of ε that trade-offs these two effects is sometimes called the *stepsize dilemma*.[57] Due to this, some heuristics and algorithms have been introduced in order to pick the value of ε,[3,35,57] Some of these methods require some a priori knowledge about the function to be differentiated, and others are based on arbitrary historical rules. If many analytical functions, like polynomials and trigonometric functions, can be computed with machine precision, numerical solutions of differential equations have errors larger than machine precision, which leads to inaccurate estimations of the gradient when ε is too small. We will further emphasize this point in Section 5.

Even with all these caveats, finite differences can be useful when computing Jacobian-vector products. Given a Jacobian matrix $J = \frac{\partial f}{\partial u}$ (or the sensitivity $s = \frac{\partial u}{\partial \theta}$) and a vector v, the product Jv corresponding to the directional derivative and can be approximated as

$$Jv \approx \frac{f(u + \varepsilon v, \theta, t) - f(u, \theta, t)}{\varepsilon} \tag{21}$$

This approach is used in numerical solvers based on Krylow methods, where linear systems are solved by iteratively solving matrix-vectors products.[38]

4.3 Complex step differentiation

An alternative to finite differences that avoids rounding errors is based on complex variable analysis. The first proposals originated in 1967 using the Cauchy integral theorem involving the numerical evaluation of a complex-valued integral.[52,53] A new approach recently emerged that uses the Taylor expansion of a function to define its complex generalization.[56,85] Assuming that we have one single scalar parameter $\theta \in \mathbb{R}$, then the function $L(\theta)$ can be expanded as the Taylor expansion

$$L(\theta + i\varepsilon) = L(\theta) + i\varepsilon L'(\theta) - \frac{1}{2}L''(\theta)\varepsilon^2 + \mathcal{O}(\varepsilon^3), \tag{22}$$

where i is the imaginary unit satisfying $i^2 = -1$. Computing the imaginary part $\mathrm{Im}(L(\theta+i\varepsilon))$ leads to

$$L'(\theta) = \frac{\mathrm{Im}(L(\theta + i\varepsilon))}{\varepsilon} + \mathcal{O}(\varepsilon^2) \tag{23}$$

The method of *complex step differentiation* consists then in estimating the gradient as $\mathrm{Im}(L(\theta + i\varepsilon))/\varepsilon$ for a small value of ε. Besides the advantage of being a method with precision $\mathcal{O}(\varepsilon^2)$, the complex step method avoids subtracting cancellation error and then the value of ε can be reduced to almost machine precision error without affecting the calculation of the derivative. Extension to higher order derivatives can be done by introducing multicomplex variables.[48]

4.4 Automatic differentiation

Automatic differentiation (AD) is a numerical method that allows computing gradients through a computer program.[33] The main idea is that every computer program manipulating numbers can be reduced to a sequence of simple algebraic operations that have straightforward derivative expressions, based upon elementary rules of differentiation. The derivatives of the outputs of the computer program with respect to their inputs are then combined using the chain rule. One advantage of AD systems is to automatically differentiate programs that include control flow, such as branching, loops or recursions. This is because any program can be reduced to a trace of input, intermediate and output variables.[5]

Depending if the concatenation of these gradients is done as we execute the program (from input to output) or in a later instance were we trace-back the calculation from the end (from output to input), we refer to *forward* or *reverse* AD, respectively.

4.4.1 Forward mode

Forward mode AD can be implemented in different ways depending on the data structures we use at the moment of representing a computer program. Examples of these data structures

include dual numbers and Wengert lists (see[5] for a good review on these methods).

Dual numbers

Dual numbers extend the definition of a numerical variable that takes a certain value to also carry information about its derivative with respect to certain parameter.[14] We can define an abstract type, defined as a dual number, composed of two elements: a *value* coordinate x_1 that carries the value of the variable and a *derivative* coordinate x_2 with the value of the derivative $\frac{\partial x_1}{\partial \theta}$. Just as complex number, we can represent dual numbers as an ordered pair (x_1, x_2), sometimes known as Argand pair, or in the rectangular form

$$x_\epsilon = x_1 + \epsilon\, x_2 \tag{24}$$

where ϵ is an abstract number called a perturbation or tangent, with the properties $\epsilon^2 = 0$ and $\epsilon \neq 0$. This last representation is quite convenient since it naturally allow us to extend algebraic operations, like addition and multiplication, to dual numbers.[45] For example, given two dual numbers $x_\epsilon = x_1 + \epsilon x_2$ and $y_\epsilon = y_1 + \epsilon y_2$, it is easy to derive using the fact $\epsilon^2 = 0$ that

$$x_\epsilon + y_\epsilon = (x_1 + y_1) + \epsilon\,(x_2 + y_2) \qquad x_\epsilon y_\epsilon = x_1 y_1 + \epsilon\,(x_1 y_2 + x_2 y_1). \tag{25}$$

From these last examples, we can see that the derivative component of the dual number carries the information of the derivatives when combining operations. For example, suppose than in the last example the dual variables x_2 and y_2 carry the value of the derivative of x_1 and x_2 with respect to a parameter θ, respectively.

Intuitively, we can think about ϵ as being a differential in the Taylor expansion:

$$\begin{aligned} f(x_1 + \epsilon x_2) &= f(x_1) + \epsilon\, x_2\, f'(x_1) + \epsilon^2 \cdot (\ldots) \\ &= f(x_1) + \epsilon\, x_2\, f'(x_1) \end{aligned} \tag{26}$$

When computing first order derivatives, we can ignore everything of order ϵ^2 or larger, which is represented in the condition $\epsilon^2 = 0$. This implies that we can use dual numbers to implement forward AD through a numerical algorithm. We will explore how this is carried in Section 5.

Computational graph

An useful way of representing a computer program is via a computational graph with intermediate variables that relate the input and output variables. Most scalar functions of interest can be represented in this factorial form as a acyclic directed graph with nodes associated to variables and edges to atomic operations,[30,33] known as Kantorovich graph[44] or Wengert trace/tape.[4,92] We can define $v_1, v_2, \ldots, v_p = \theta_1, \theta_2, \ldots, \theta_p$ the input set of variables; $v_{p+1}, \ldots, v_{m-1}$ the set of all the intermediate variables, and finally $v_m = L(\theta)$ the final output of a computer program. This can be done in such a way that the order is strict, meaning that each variable v_i is computed just as a function of the previous variables v_j with $j < i$. Once the graph is constructed, we can compute the derivative of every node with respect to other (a quantity known as the tangent) using Bauer formula[4,65]

$$\frac{\partial v_j}{\partial v_i} = \sum_{\substack{\text{paths } w_0 \to w_1 \to \ldots \to w_K \\ \text{with } w_0 = v_i, w_K = v_j}} \prod_{k=0}^{K-1} \frac{\partial w_{k+1}}{\partial w_k}, \tag{27}$$

where the sum is calculated with respect to all the directed paths in the graph connecting the input and target node. Instead of evaluating the last expression for all possible path, a simplification is to increasingly evaluate $j = p + 1, \ldots, m$ using the recursion

$$\frac{\partial v_j}{\partial v_i} = \sum_{w \text{ such that } w \to v_j} \frac{\partial v_j}{\partial w} \frac{\partial w}{\partial v_i} \tag{28}$$

Since every variable node w such that $w \to v_j$ is an edge of the computational graph have index less than j, we can iterate this procedure as we run the computer program and solve for both the function and its gradient. This is possible because in forward mode the term $\frac{\partial w}{\partial v_i}$ has been computed in a previous iteration, while $\frac{\partial v_j}{\partial w}$ can be evaluated at the same time the node v_j is computed based on only the value of the parent variable nodes. The only requirement for differentiation is being able to compute the derivative/tangent of each edge/primitive and combine these using the recursion (28).

4.4.2 Backward mode

Backward mode AD is also known as the adjoint of cotangent linear mode or backpropagation in the field of machine learning. The reverse mode of automatic differentiation has been introduced in different contexts[31] and materializes the observation made by Phil Wolfe that if the chain rule is implemented in reverse mode, then the ratio between the computation of the gradient of a function and the function itself can be bounded by a constant that do not depend of the number of parameters to differentiate,[30,94] a point known as *cheap gradient principle*.[31] Given a directional graph of operations defined by a Wengert list,[92] we can compute gradients of any given function in the same fashion as Equation (28) but in backwards mode as

$$\bar{v}_i = \frac{\partial \ell}{\partial v_i} = \sum_{w : v \to w \in G} \frac{\partial w}{\partial v} \bar{w}. \tag{29}$$

In this context, the notation $\bar{\omega} = \frac{\partial \ell}{\partial \omega}$ is introduced to signify the partial derivative of the final loss function with respect to various program variables. This derivative is often referred to as the adjoint or cotangent, though it should not be confused with the adjoint method discussed in later sections.

Since in backwards AD the values of $\bar{\omega}$ are being updated in reverse order, in order to evaluate the terms $\frac{\partial \omega}{\partial v}$ we need to know the state value of all the argument variables v of ω, which need to be stored in memory during the evaluation of the function in order to be able to apply backward AD.

Another way of implementing backwards AD is by defining a *pullback* function,[36] a method also known as *continuation-passing style*.[89] In the backward step, this executes a series of function calls, one for each elementary operation. If one of the nodes in the graph w is the output of an operation involving the nodes $v_1, \ldots, v_m$, where $v_i \to w$ are all nodes in the graph, then the pullback $\bar{v}_1, \ldots, \bar{v}_m = \mathcal{B}_w(\bar{w})$ is a function that accepts gradients with respect to w (defined as $\bar{w}$) and returns gradients with respect to each v_i ($\bar{v}_i$) by applying the chain rule. Consider the example of the multiplicative operation $w = v_1 \times v_2$. Then

$$\bar{v}_1, \bar{v}_2 = v_2 \times \bar{w}, \quad v_1 \times \bar{w} = \mathcal{B}_w(\bar{w}), \tag{30}$$

which is equivalent to using the chain rule as

$$\frac{\partial \ell}{\partial v_1} = \frac{\partial}{\partial v_1}(v_1 \times v_2)\frac{\partial \ell}{\partial w}. \tag{31}$$

4.4.3 AD connection with JVPs and VJPs

When working with unit operations that involve matrix operations dealing with vectors of different dimensions, the order in which we apply the chain rule matters.[26] When computing a gradient using AD, we can encounter vector-Jacobian products (VJPs) or Jacobian-vector products (JVP). As their name indicates, the difference between them regards the fact that the quantity we are interested in computing is described by the product of a Jacobian by a vector on the left side (VJP) or the right (JVP).

For nested functions, the Jacobian is described as the product of multiple Jacobian using the chain rule. In this case, the full gradient is computed as the chain product of vectors and Jacobians. Let us consider for example the case of a loss function $L : \mathbb{R}^p \mapsto \mathbb{R}$ that can be decomposed as $L(\theta) = \ell \circ g_k \circ \ldots \circ g_2 \circ g_1(\theta)$, with $\ell : \mathbb{R}^{d_k} \mapsto \mathbb{R}$ the final evaluation of the loss function after we apply in order a sequence of intermediate functions $g_i : \mathbb{R}^{d_{i-1}} \mapsto \mathbb{R}^{d_i}$, $d_0 = p$. Now, using the chain rule, we can calculate the gradient of the final loss function as

$$\nabla_\theta L = \nabla \ell \cdot Dg_k \cdot Dg_{k-1} \cdot \ldots \cdot Dg_2 \cdot Dg_1, \tag{32}$$

with Dg_i the Jacobians of each nested function evaluated at ... Notice that in the last equation, $\nabla \ell \in \mathbb{R}^{d_k}$ is a vector. In order to compute $\nabla_\theta L$, we can solve the multiplication starting from the right side, which will correspond to multiply the Jacobians forward from Dg_1 to Dg_k, or from the left side, moving backwards. The important aspect of this last case is that we will always be computing VJPs, since $\nabla \ell \in \mathbb{R}^{d_k}$ is a vector. Since VJPs are easier to evaluate than full Jacobians, the backward mode will be in general faster (see Figure 2). For general rectangular matrices $A \in \mathbb{R}^{d_1 \times d_2}$ and $B \in \mathbb{R}^{d_2 \times d_3}$, the cost of the matrix multiplication AB is $\mathcal{O}(d_1 d_2 d_3)$ (more efficient algorithms exist but this does not impact these results). This implies that forward AD requires a total of

$$d_2 d_1 n + d_3 d_2 p + \ldots + d_k d_{k-1} p + d_k p = \mathcal{O}(p) \tag{33}$$

operations, while backwards mode AD requires

$$d_k d_{k-1} + d_{k-1} d_{k-2} + \ldots + d_2 d_1 + d_1 n = \mathcal{O}(1) \tag{34}$$

operations, where the $\mathcal{O}$ is with respect to the variable p.

When the function to differentiate has a larger input space than output, AD in backward mode is more efficient as it propagates the chain rule by computing VJPs, the reason why backwards AD is more used in modern machine learning. However, notice that backwards mode AD requires us to save the solution through the forward run in order to run backwards afterwards,[6] while in forward mode we can just evaluate the gradient as we iterate our sequence of functions. We discuss in section XXX how this problem can be overcome with a good checkpointing scheme. This means that for problems with a small number of parameters, forward mode can be faster and more memory-efficient that backwards AD.

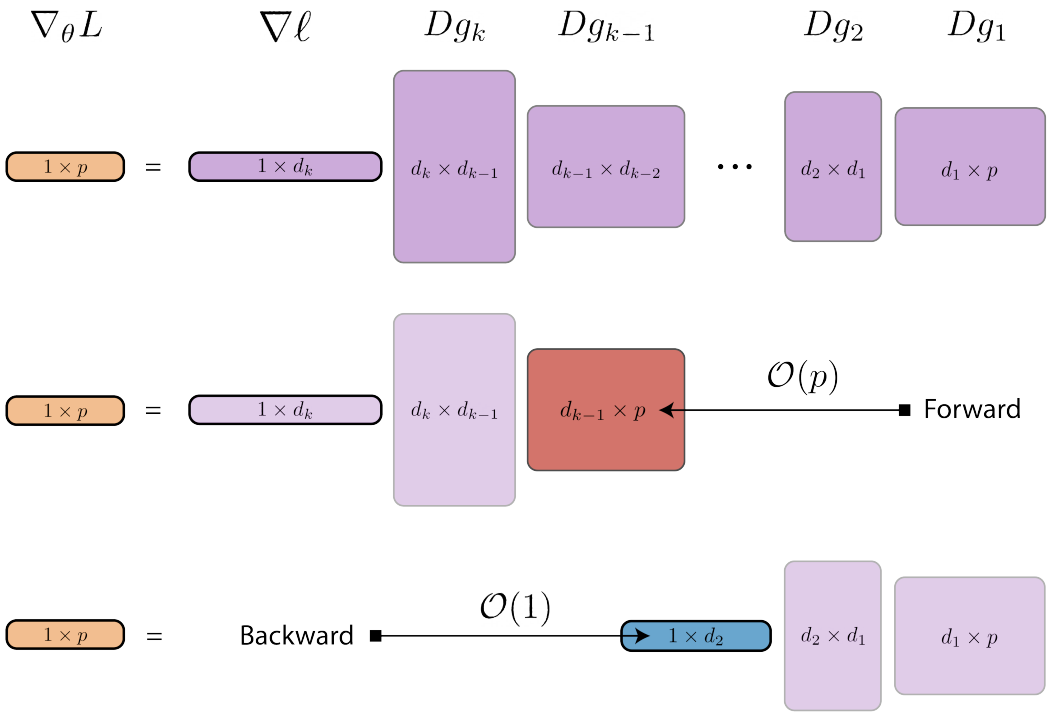

Figure 2: Comparison between forward and backward AD. Changing the order of Jacobian multiplications changes the total number of floating-point operations, which leads to different computational complexities between forward and backward mode. However, backwards mode requires storing in memory the value of each computational operation during the forward pass, while forward mode can update the gradient on running time.

4.5 Symbolic differentiation

not sure this deserves a full section

Exponentially large symbolic expressions[5]

Sometimes AD is compared against symbolic differentiation. According to,[49] these two are the same and the only difference is in the data structures used to implement them, while[19] suggests that AD is symbolic differentiation performed by a compiler.

4.6 Sensitivity equations

An easy way to derive an expression for the sensitivity s is by deriving the sensitivity equations,[70] a method also referred to as continuous local sensitivity analysis (CSA). If we consider the original system of ODEs and we differentiate with respect to θ, we then obtain

$$\frac{d}{d\theta}\frac{du}{dt} = \frac{d}{d\theta} f(u(\theta), \theta, t) = \frac{\partial f}{\partial \theta} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial \theta}, \tag{35}$$

that is

$$\frac{ds}{dt} = \frac{\partial f}{\partial u} s + \frac{\partial f}{\partial \theta}. \tag{36}$$

By solving the sensitivity equation at the same time we solve the original differential equation for $u(t)$, we ensure that by the end of the forward step we have calculated both $u(t)$ and $s(t)$. This also implies that as we solve the model forward, we can ensure the same level of numerical precision for the two of them.

In opposition to the methods previously introduced, the sensitivity equations find the gradient by solving a new set of continuous differential equations. Notice also that the obtained sensitivity $s(t)$ can be evaluated at any given time t. This method can be labeled as forward, since we solve both $u(t)$ and $s(t)$ as we solve the differential equation forward in time, without the need of backtracking any operation though the solver.

For systems of equations with few number of parameters, this method is useful since the system of equations composed by Equations (8) and (36) can be solved in $\mathcal{O}(np)$ using the same precision for both solution and sensitivity numerical evaluation. Furthermore, this method does not required saving the solution in memory, so it can be solved purely in forward mode without backtracking operations. However, notice that the term $\frac{\partial f}{\partial u}s$ is in general difficult to compute.

It is important to remark that the sensitivity equations can be also solved in discrete forward mode by numerically discretizing the original ODE and later deriving the discrete sensitivity equations. For most cases, this leads to the same result that in the continuous case.[96]

4.7 Adjoint methods

For complex and large systems, computing the gradient directly on top of the numerical solver (for example, using AD) can be memory expensive since the large number of function evaluations required by the solver and the later store of the intermediate states. For these cases, the adjoint-based method allows to compute the gradients of a loss function by instead computing an intermediate variable (the adjoint) that serves as a bridge between the solution of the ODE and the final sensitivity.

There is a large family of adjoint methods that in first order can be classified as discrete and continuous adjoints. The former usually arises as the numerical discretization of the latter, and in general, these two give different different computational results.[83] Different results exist regarding the consistency or inconsistency between the two approaches, and this usually depends on the ODE and equation. Proofs of the consistency of discrete adjoint methods for Runge-Kutta methods have been provided in.[77,78] Depending on the choice of the Runge-Kutta coefficients, we can have a numerical scheme that is both consistent for the original equation and consistent/inconsistent for the adjoint.[34] Furthermore, adjoint methods can fail in chaotic systems.[90]

4.7.1 Discrete adjoint method

Also known as the adjoint state method, it is another example of a discrete method that aims to find the gradient by solving an alternative system of linear equations, known as the *adjoint equations*, at the same time that the original system of linear equations defined by the numerical solver is solved. These methods are extremely popular in optimal control theory in fluid dynamics, for example for the design of geometries for vehicles and airplanes that

optimize performance.[20,27] This approach follows the discretize-optimize approach, meaning that we first discretize the system of continuous ODEs and then solve on top of these linear equations.[27]

Discrete differential equation

The derivation of the discrete adjoint equations is carried once the numerical scheme for solving Equation (8) has been specified. Given a discrete sequence of timesteps $t_0, t_1, \ldots, t_N$, we evaluate the solution at $u_i = u(t_i; \theta)$. Some of the most common numerical solvers include multistep linear solvers of the form

$$\sum_{i=0}^{K_1} \alpha_{ni} u_{n-i} + h_n \sum_{i=0}^{K_2} \beta_{ni} f(u_{n-i}, \theta, t_{n-i}) = 0. \tag{37}$$

and Runge-Kutta methods with

$$u_{n+1} = u_n + h \sum_{i=1}^{s} b_i k_i \tag{38}$$

$$k_i = f\left(u_n + \sum_{j=1}^{s} a_{ij} k_j, \theta, t_n + c_n h\right) \qquad i = 1, 2, \ldots, s. \tag{39}$$

The former is linear in f, which for example is not the case in Runge-Kutta methods with intermediate evaluations.[1] Explicit methods are characterized by $\beta_{n,0} = 0$ for the multistep and $a_{ij} = 0$ if $i \leq j$ for Runge-Kutta methods, otherwise, the method is implicit.

For multistep methods, solving the differential equation implies to be able to solve the system of constraints

$$g_i(u_i; \theta) = u_i - h\beta_{n0} f(u_i, \theta, t_i) - \alpha_i = 0 \tag{40}$$

where α_i has includes the information of all the past iterations. This system can be solved sequentially, by solving for u_i in increasing order of index using Newton method. If we call the super-vector $U = (u_1, u_2, \ldots, u_N) \in \mathbb{R}^{nN}$, we can combine all these equations into one single system of equations $G(U) = (g_1(u_1; \theta), \ldots, g_N(u_N; \theta)) = 0$.

In the simplest case where the algebraic set of equations is linear and we can write $g_i(u_{i+1}; \theta) = u_{i+1} - A_i(\theta)\, u_i - b_i$ with $A_i \in \mathbb{R}^{n \times n}$ and $b_i \in \mathbb{R}^n$ defined by the numerical solver, the condition $G(U) = 0$ simplifies to the linear system of equations

$$A(\theta)U = \begin{bmatrix} \mathbb{I}_{n\times n} & 0 & & & \\ -A_1 & \mathbb{I}_{n\times n} & 0 & & \\ & -A_2 & \mathbb{I}_{n\times n} & 0 & \\ & & & \ddots & \\ & & & -A_{N-1} & \mathbb{I}_{n\times n} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} A_0 u_0 + b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-1} \end{bmatrix} = b(\theta), \tag{41}$$

with $\mathbb{I}_{n\times n}$ the identity matrix of size $n \times n$. It is important to notice that in most cases, the matrix $A(\theta)$ is quite large and mostly sparse. If this representation of the discrete differential

equation is quite convenient for mathematical manipulations, at the moment of solving the system we will rely in iterative solvers that save memory and computation.

Adjoint state equations

We are interested in differentiating a function $L(U, \theta)$ with respect to the parameter θ. Since here U is the discrete set of evaluations of the solution, examples of loss functions now include

$$L(U, \theta) = \frac{1}{2} \sum_{i=1}^{N} \|u_i - u_i^{\text{obs}}\|^2, \tag{42}$$

with u_i^{obs} the observed time-series. We further need to impose the constraint that the solution satisfies the algebraic linear equation $G(U; \theta) = 0$. Now,

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial U} \frac{\partial U}{\partial \theta}, \tag{43}$$

and also for the constraint $G(U; \theta) = 0$ we can derive

$$\frac{dG}{d\theta} = \frac{\partial G}{\partial \theta} + \frac{\partial G}{\partial U} \frac{\partial U}{\partial \theta} = 0 \tag{44}$$

which is equivalent to

$$\frac{\partial U}{\partial \theta} = - \left(\frac{\partial G}{\partial U} \right)^{-1} \frac{\partial G}{\partial \theta}. \tag{45}$$

If we replace this last expression into equation (43), we obtain

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \underbrace{\frac{\partial L}{\partial U}}_{\text{vector}} \left(\frac{\partial G}{\partial U} \right)^{-1} \frac{\partial G}{\partial \theta}. \tag{46}$$

The important trick in the adjoint state methods is to observe that in this last equation, the right-hand side can be resolved as a vector-Jacobian product (VJP). Instead of computing the product of the matrices $\left(\frac{\partial G}{\partial U}\right)^{-1}$ and $\frac{\partial G}{\partial \theta}$, it is computationally more efficient first to compute the resulting vector from the operation $\frac{\partial L}{\partial U} \left(\frac{\partial G}{\partial U}\right)^{-1}$ and then multiply this by $\frac{\partial G}{\partial \theta}$. This is what leads to the definition of the adjoint $\lambda \in \mathbb{R}^{nN}$ as the solution of the linear system of equations

$$\left(\frac{\partial G}{\partial U} \right)^T \lambda = \left(\frac{\partial L}{\partial U} \right)^T, \tag{47}$$

that is,

$$\lambda^T = \frac{\partial L}{\partial U} \left(\frac{\partial G}{\partial U} \right)^{-1}. \tag{48}$$

Finally, if we replace Equation (48) into (46), we obtain

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \lambda^T \frac{\partial G}{\partial \theta}. \tag{49}$$

The important trick to notice here is the rearrangement of the multiplicative terms involved in equation (46). Computing the full Jacobian/sensitivity $\partial u/\partial\theta$ will be computationally expensive and involves the product of two matrices. However, we are not interested in the calculation of the Jacobian, but instead in the VJP given by $\frac{\partial L}{\partial U}\frac{\partial U}{\partial \theta}$. By rearranging these terms, we can make the same computation more efficient.

Notice that the algebraic equation of the adjoint λ in Equation (47) is a linear system of equations even when the original system $G(U) = 0$ was not necessarily linear in U. For the linear system of discrete equations $G(U;\theta) = A(\theta)U - b(\theta) = 0$, we have[41]

$$\frac{\partial G}{\partial \theta} = \frac{\partial A}{\partial \theta}U - \frac{\partial b}{\partial \theta}, \tag{50}$$

so the desired gradient in Equation (49) can be computed as

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \lambda^T\left(\frac{\partial A}{\partial \theta}U - \frac{\partial b}{\partial \theta}\right) \tag{51}$$

with λ the solution of the linear system (Equation (47))

$$A(\theta)^T\lambda = \begin{bmatrix} \mathbb{I}_{n\times n} & -A_1^T & & & \\ 0 & \mathbb{I}_{n\times n} & -A_2^T & & \\ & 0 & \mathbb{I}_{n\times n} & -A_3^T & \\ & & & \ddots & -A_{N-1}^T \\ & & & 0 & \mathbb{I}_{n\times n} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \vdots \\ \lambda_N \end{bmatrix} = \begin{bmatrix} u_1 - u_1^{\text{obs}} \\ u_2 - u_2^{\text{obs}} \\ u_3 - u_3^{\text{obs}} \\ \vdots \\ u_N - u_N^{\text{obs}} \end{bmatrix} = \frac{\partial L}{\partial U}^T. \tag{52}$$

This is a linear system of equations with the same size of the original $A(\theta)U = b(\theta)$, but involving the adjoint matrix A^T. Computationally this also means that if we can solve the original system of discretized equations then we can also solve the adjoint. One way of doing this is relying on matrix factorization. Using the LU factorization we can write the matrix $A(\theta)$ as the product of a lower and upper triangular matrices $A(\theta) = LU$, which then can be also used for solving the adjoint equation since $A^T(\theta) = U^T L^T$. Another more natural way of finding the adjoints λ is by noticing that the system of equations (52) is equivalent to the iterative scheme

$$\lambda_i = A_i^T\lambda_{i+1} + (u_i - u_i^{\text{obs}}) \tag{53}$$

with initial condition λ_N. This means that we can solve the adjoint equation in backwards mode, starting from the final state λ_N and computing the values of λ_i in decreasing index order. Notice that this procedure requires to know the value of u_i at any given timestep.

4.7.2 Continuous adjoint method

The continuous adjoint method, also known as continuous adjoint sensitivity analysis (CASA), operates by defining a convenient set of new differential equations for the adjoint variable and using this to compute the gradient in a more efficient manner. Mathematically speaking, the adjoint equations can be derived from a duality or Lagrangian point of view.[27] We prefer to derive the equation using the former methods since we believe it gives better insights to how the method works and also allow to generalize to other user cases. The derivation of

both the discrete and continuous adjoint methods using Lagrangian multipliers can be found in Appendix A. We encourage the interested reader to make the effort of following how the continuous adjoint method follows the same logic than the discrete methods, but where the discretization of the differential equation does not happen until the very last step, when the solutions of the differential equations involved need to be numerically evaluated.

Consider an integrated loss function of the form

$$L(u;\theta) = \int_{t_0}^{t_1} h(u(t;\theta),\theta)dt \tag{54}$$

and its derivative with respect to the parameter θ given by the following integral involving the sensitivity matrix $s(t)$:

$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u}s(t)\right) dt. \tag{55}$$

Just as in the case of the discrete adjoint method, the complicated term to evaluate in the last expression is the sensitivity (Equation (18)). Just as in the case of the discrete adjoint method, the trick is to evaluate the VJP $\frac{\partial h}{\partial u}\frac{\partial u}{\partial \theta}$ by defining an intermediate adjoint variable. The continuous adjoint equation now is obtained by finding the dual/adjoint equation of the sensitivity equation using the weak formulation of Equation (36). The adjoint equation is obtained by writing the sensitivity equation in the form

$$\int_{t_0}^{t_1} \lambda(t)^T \left(\frac{ds}{dt} - f(u,\theta,t)\, s - \frac{\partial f}{\partial \theta}\right) dt = 0, \tag{56}$$

where this equation must be satisfied for every function $\lambda(t)$ in order for Equation (69) to be true. The next step is to get rid of all time derivative applied to the sensitivity $s(t)$ using integration by parts:

$$\int_{t_0}^{t_1} \lambda(t)^T \frac{ds}{dt} dt = \lambda(t_1)^T s(t_1) - \lambda(t_0)^T s(t_0) - \int_{t_0}^{t_1} \frac{d\lambda^T}{dt} s(t)\, dt. \tag{57}$$

Replacing this last expression into Equation (56) we obtain

$$\int_{t_0}^{t_1} \left(-\frac{d\lambda^T}{dt} - \lambda(t)^T f(u,\theta,t)\right) s(t) dt = \int_{t_0}^{t_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt - \lambda(t_1)^T s(t_1) + \lambda(t_0)^T s(t_0). \tag{58}$$

At first glance, there is nothing particularly interesting about this last equation. However, both Equations (55) and (58) involved a VJP with $s(t)$. Since Equation (58) must hold for every function $\lambda(t)$, we can pick $\lambda(t)$ to make the terms involving $s(t)$ in Equations (55) and (58) to perfectly coincide. This is done by defining the adjoint $\lambda(t)$ to be the solution of the new system of differential equations

$$\frac{d\lambda}{dt} = -f(u,\theta,t)^T \lambda - \frac{\partial h^T}{\partial u} \qquad \lambda(t_1) = 0. \tag{59}$$

Notice that the adjoint equation is defined with the final condition at t_1, meaning that it needs to be solved backwards in time. The definition of the adjoint $\lambda(t)$ as the solution of this last ODE simplifies Equation (58) to

$$\int_{t_0}^{t_1} \frac{\partial h}{\partial u} s(t) dt = \lambda(t_0)^T s(t_0) + \int_{t_0}^{t_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt. \tag{60}$$

Finally, replacing this inside the expression for the gradient of the loss function we have

$$\frac{dL}{d\theta} = \lambda(t_0)^T s(t_0) + \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta} \right) dt \tag{61}$$

The full algorithm to compute the full gradient $\frac{dL}{d\theta}$ can be described as follows:

1. Solve the original differential equation $\frac{du}{dt} = f(u, t, \theta)$;
2. Solve the backwards adjoint differential equation (59);
3. Compute the gradient using Equation (61).

5 Numerical implementation

In this section, we address how these different methods are computationally implemented and how to decide which method to use depending on the scientific task. In order to address this point, it is important to make one further distinction of the methods introduced in Section 4 between those that apply direct differentiation at the algorithmic level or those that are based on numerical solvers. The first is easier to implement since they are agnostic with respect to the details of the ODE and its numerical solution; however, they tend to be either inaccurate, memory-expensive, or unfeasible for large models. The family of methods that are based on numerical solvers include the sensitivity equations and the adjoint methods, both discrete and continuous; they are more difficult to implement and for real case applications require complex software implementations, but they are also more efficient.

5.1 Direct methods

Direct methods are implemented independently of the structure of the ODE and the numerical solver used to solve it.

Finite differences are easy to implement manually, do not require much software support, and provide a direct way of approximating a gradient. In Julia, these methods are implemented in `FiniteDiff.jl` and `FiniteDifferences.jl` and it is recommended to use establish libraries than implementing it yourself, since these already include subroutines to determine step-sizes. However, finite differences are less accurate (see section XXX) and as costly as forward AD[30] and complex-step differentiation. Figure 3 illustrates the error in computing the gradient of a simple loss function for both true analytical solution and numerical solution of a system of ODEs as a function of the stepsize ε using finite differences and complex-step differentiation. The error when using a numerical solver is larger and it is dependent on the numerical precision of the numerical solver.

There is sort of a contradiction between these two sen-

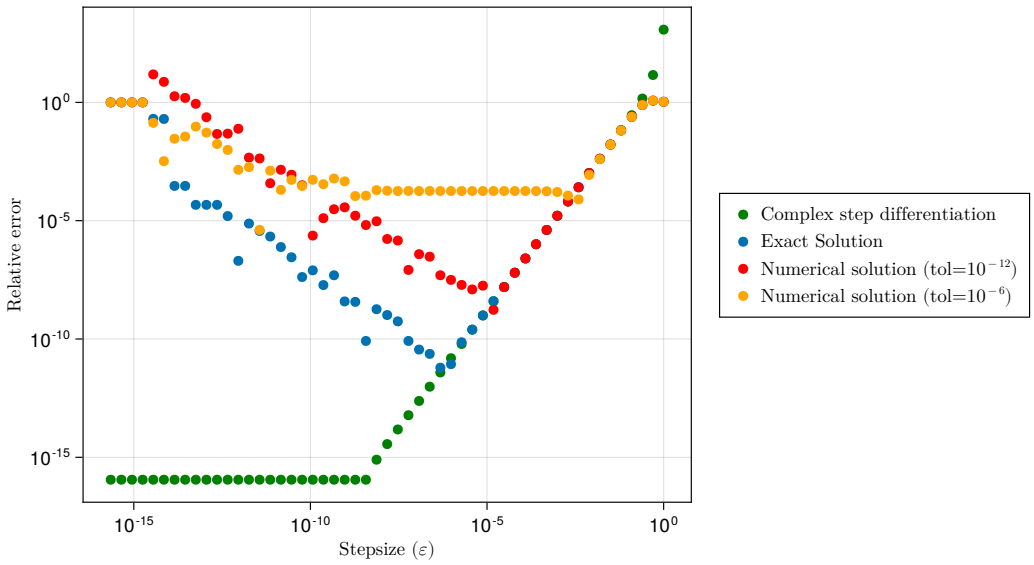

Figure 3: Absolute relative error when computing the gradient of the function $u(t) = \sin(\omega t)/\omega$ with respect to ω at $t = 10.0$ as a function of the stepsize ε. Here $u(t)$ corresponds to the solution of the differential equation $u'' + \omega^2 u = 0$ with initial condition $u(0) = 0$ and $u'(0) = 1$. The blue dots correspond to the case where the relative error is computed with finite differences. The red and orange lines are for the case where $u(t)$ is numerically computed using the default Tsitouras solver[87] from `OrdinaryDiffEq.jl` using different tolerances. The error when using a numerical solver is larger and it is dependent on the numerical precision of the numerical solver.

Implementing forward AD using dual numbers is usually carried out using *operator overloading*.[62] This means expanding the object associated to a numerical value to include its dual components (derivative) and extending the definition of atomic algebraic functions. In Julia, this can be done by relying on multiple dispatch. The following example illustrates how to define a dual number and its associated binary addition and multiplication extensions.

```
using Base: @kwdef

@kwdef struct DualNumber{F <: AbstractFloat}
    value::F
    derivative::F
end

# Binary sum
Base.:(+)(a::DualNumber, b::DualNumber) = DualNumber(value = a.value + b.value,
    derivative = a.derivative + b.derivative)

# Binary product
Base.:(*)(a::DualNumber, b::DualNumber) = DualNumber(value = a.value * b.value,
    derivative = a.value*b.derivative + a.derivative*b.value)
```

We further overload base operations for this new type to extend the definition of standard functions by simply applying the chain rule and storing the derivative in the dual variable

following Equation (26):

```
function Base.:(sin)(a::DualNumber)
    value = sin(a.value)
    derivative = a.derivative * cos(a.value)
    return DualNumber(value=value, derivative=derivative)
end
```

In the Julia ecosystem, `ForwardDiff.jl` implements forward mode AD with multidimensional dual numbers.[73] Notice that a major limitation of the dual number approach is that a dual variable is required for each variable to differentiate. Incorrect implementations of this aspect can lead to *perturbation confusion*,[55,84] an existing problem in some AD software where dual variables corresponding to different variables become indistinguishable, specially in the case of nested functions.[55]

Implementations of forward AD using dual numbers and computational graphs require a number of operations that increases with the number of variables to differentiate, since each computed quantity is accompanied by the corresponding gradient calculations.[30] This consideration also applies to the other forward methods, including finite differences and complex-step differentiation, which makes forward models inefficient when differentiating with respect to many parameters.

Notice that both AD based on dual number and complex-step differentiation introduce an abstract unit (ϵ and i, respectively) associated with the imaginary part of the extender value that carries forward the numerical value of the gradient. Although these methods seem similar, it is important to remark that AD gives the exact gradient, while complex step differentiation relies on numerical approximations that are valid just when the stepsize ε is small. The next example shows how the calculation of the gradient of $\sin(x^2)$ is performed by these two methods:

can we talk about the curse of dimensionality?

Operation	AD with Dual Numbers	Complex Step Differentiation
x	$x + \epsilon$	$x + i\varepsilon$
x^2	$x^2 + \epsilon\,(2x)$	$x^2 - \varepsilon^2 + 2i\varepsilon x$
$\sin(x^2)$	$\sin(x^2) + \epsilon\,\cos(x^2)(2x)$	$\sin(x^2 - \varepsilon^2)\cosh(2i\varepsilon) + i\,\cos(x^2 - \varepsilon^2)\sinh(2i\varepsilon)$

(62)

While the second component of the dual number has the exact derivative of $\sin(x^2)$, it is not until we take $\varepsilon \to 0$ than we obtain the derivative in the imaginary component for the complex step method

$$\lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \cos(x^2 - \varepsilon^2)\sinh(2i\varepsilon) = \cos(x^2)(2x). \tag{63}$$

The dependence of the complex step differentiation method on the step size gives it a closer resemblance to finite difference methods than to automatic differentiation (AD) using dual numbers. This difference between the methods also makes the complex step method sometimes more efficient than both finite differences and AD,[48] an effect that can be counterbalanced by the number of extra unnecessary operations that complex arithmetic requires (see

last column of (62)).[56] It is also important to remark that many modern software already have support for complex number arithmetic, making complex step differentiation very easy to implement.

The libraries `ReverseDiff.jl` and `Zygote.jl` use callbacks to compute gradients. When gradients are being computed with less than ~ 100 parameters, the former is faster (see documentation).

[5]

Notice that the application of reverse AD on a numerical solver (without checkpointing) scales as $\mathcal{O}(nk)$, with k the number of steps of the numerical solver. Furthermore, when reverse AD is applied on the numerical solver, the step-size needs to be adapted to ensure the stability of the backward stesps, further increasing the computational complexity.

5.1.1 Further remarks

A crucial distinction between AD implementations based on computational graphs is between static and dynamical graphs.[5]

5.2 Solver-based methods

Sensitivity methods based on numerical solvers tend to be better adapted to the structure and properties of the underlying ODE (stiffness, stability, accuracy) but are also more difficult to implement. This difficulty arises from the fact that the sensitivity method needs to deal with some numerical and computational considerations, including how to handle matrix/Jacobian-vector products; numerical stability of the forward/backward solver; and memory-time tradeoff. These factors are further exacerbated by the number of ODEs and parameters in the model. Just a few modern scientific software have the capabilities of handling ODE solvers and computing their sensitivities at the same time. These include `CVODES` within `SUNDIALS` in C;[35,81] `ODESSA`[50] and `FATODE` (discrete adjoints)[96] both in Fortram; `SciMLSensitivity.jl` in Julia;[68] `Dolfin-adjoint` based on the `FEniCS` Project.[21,58]

It is important to remark that the underlying machinery of all solvers relies on solvers for linear systems of equations, which can be solved in dense, band (sparse), and Krylow mode. Another important consideration is that all these methods have subroutines to compute the VJPs involved in the sensitivity and adjoint equations. This calculation is carried out by another sensitivity method (finite differences, AD) which plays a central role when analyzing the accuracy and stability of the adjoint method.

5.2.1 Sensitivity equation

5.2.2 Solving the adjoint

An equally important consideration when working with adjoints is when these are numerically stable. Some works have shown that continuous adjoints can lead to unstable sensitivities.[40] Implicit forward schemes can give rise to explicit backwards schemes, leading to unstable solutions for the gradient.

Repetition

We need to define what are callbacks

I think that we should introduce checkpointing before

seems like a repetition

not clear

Solving the backwards mode

The bottleneck of this method is the calculation of the adjoint since in order to solve the adjoint equation we need to know $u(t)$ at any given time. Effectively, notice that the adjoint equation involves the terms $f(u, \theta, t)$ and $\frac{\partial h}{\partial u}$ which are both functions of $u(t)$. There are different ways of addressing the evaluation of $u(t)$ during the backwards step.

(i) **Dense Store.** During the forward model, we can just store in memory all the intermediate states of the numerical solution. This leads to heavy-memory expensive algorithms.

(ii) **Re-solve.** Solve again the original ODE together with the adjoint as the solution of the reversed augmented system[13]

$$\frac{d}{dt}\begin{bmatrix} u \\ \lambda \\ \frac{dL}{d\theta} \end{bmatrix} = \begin{bmatrix} -f \\ -\frac{\partial f}{\partial u}^T \lambda - \frac{\partial h}{\partial u}^T \\ -\lambda^T \frac{\partial f}{\partial \theta} - \frac{\partial h}{\partial \theta} \end{bmatrix} \qquad \begin{bmatrix} u \\ \lambda \\ \frac{dL}{d\theta} \end{bmatrix}(t_1) = \begin{bmatrix} u(t_1) \\ \frac{\partial L}{\partial u(t_1)} \\ \lambda(t_0)^T s(t_0) \end{bmatrix}. \tag{64}$$

However, computing the ODE backwards can be unstable and lead to large numerical errors.[47,97]

(iii) **Checkpointing.** Also known as windowing, checkpointing is a technique that tradeoffs memory and time by saving intermediate states of the solution in the forward pass and recalculating the solution between intermediate states in the backwards mode.[33,80] This is implemented in `Checkpointing.jl`.[80]

One way of solving this system of equations that ensures stability is by using implicit methods. However, this requires cubic time in the total number of ordinary differential equations, leading to a total complexity of $\mathcal{O}((n+p)^3)$ for the adjoint method. Two alternatives are proposed in,[47] the first referred to as *Quadrature Adjoint* produces a high order interpolation of the solution $u(t)$ as we move forward, then solve for λ backwards using an implicit solver and finally integrating $\frac{dL}{d\theta}$ in a forward step. This reduces the complexity to $\mathcal{O}(n^3+p)$, where the cubic cost in the number of ODEs comes from the fact that we still need to solve the original stiff differential equation in the forward step. A second but similar approach is to use an implicit-explicit (IMEX) solver, where we use the implicit part for the original equation and the explicit for the adjoint. This method also has a complexity of $\mathcal{O}(n^3+p)$.

Solving the quadrature

Another computational consideration is how the integral in Equation (61) is numerically evaluated. Some methods save computation by noticing that the last step in the continuous adjoint method of evaluating $\frac{dL}{d\theta}$ is an integral instead of an ODE, and then can be evaluated as such without the need to include it in the tolerance calculation inside the numerical solver.[46] Numerical integration, also known as quadrature integration, consists in approximating integrals by finite sums of the form Numerical solutions of the integral

$$\int_{t_0}^{t_1} F(t)dt \approx \sum_{i=1}^{K} \omega_i \, F(\tau_i), \tag{65}$$

where the evaluation of the function occurs in certain knots $t_0 \leq \tau_1 < \ldots < \tau_K \leq t_1$, and ω_i are weights. Weights and knots are obtained in order to maximize the order in which polynomials are exactly integrated.[86]

Different quadrature methods are based on different choices of the knots and associated weights. Between these methods, the Gaussian quadrature is the faster method to evaluate one-dimensional integrals.[63]

5.2.3 Computing VJPs

All the methods analized in this section need to deal with the calculation of VJPs. The choice of the specific algorithm to compute VJPs can have significant impact in the overall performance of the sensitivity method.

In SUNDIALS, the VJPs involved in the sensitivity and adjoint method are handled using finite differences unless specified by the user.[35] In FATODE, these can be computed with finite differences, AD or provides by the user.

In the Julia ecosystem, different AD packages are available for this task, including `ForwardDiff.jl`, `ReverseDiff.jl`, `Zygote.jl`,[37] `Enzyme.jl`,[60] `Tracker.jl`. Each one of these have different advantages depending on scientific problem, computational features, and hardware resources. This lead to the development of `AbstractDifferentiation.jl` which allows to combine different methods.[79]

6 Recommendations

For sufficient small systems of less than 100 parameters and ODEs, Forward AD is the most efficient method, outperforming sensitivity and adjoint methods.[54]

We should pay attention that this section does not overlap with.[54]

Practitioners would surely benefit from recommendations on which type of optimizer to use, in the case of inverse modelling. For instance, it is useful to know that Adam is much less computationally demanding than BFGS, and is less prone to converging to local minimum. Adam + BFGS is a nice combination to avoid local minima while obtaining a precise parameter estimation. An experiment which shows how e.g. SDG, Adam, Radam, BFGS and LBFGS perform on the minimzation of a loss involving different models, in terms of convergence vs computational complexity. I would be happy to provide such an experiment.

Appendices

A Lagrangian derivation of adjoints

In this section we derive the adjoint equation for both discrete and continuous methods using the Lagrange multiplier trick. Conceptually, the method is the same in both discrete and continuous case, with the difference that we manipulate linear algebra objects for the former and continuous operators for the later.

For the continuous adjoint method, we proceed the same way by writing a new loss function $I(\theta)$, sometimes known as the *Lagrangian*, identical to $L(\theta)$ as

$$I(\theta) = L(\theta) - \int_{t_0}^{t_1} \lambda(t)^T \left(\frac{du}{dt} - f(u, \theta, t) \right) dt \tag{66}$$

where $\lambda(t) \in \mathbb{R}^n$ is the Lagrange multiplier of the continuous constraint defined by the differential equation. Now,

$$\frac{dL}{d\theta} = \frac{dI}{d\theta} = \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u}\frac{\partial u}{\partial \theta} \right) dt - \int_{t_0}^{t_1} \lambda(t)^T \left(\frac{d}{dt}\frac{du}{d\theta} - \frac{\partial f}{\partial u}\frac{du}{d\theta} - \frac{\partial f}{\partial \theta} \right) dt. \tag{67}$$

Notice that the term involved in the second integral is the same we found when deriving the sensitivity equations. We can derive an easier expression for the last term using integration by parts. Using our usual definition of the sensitivity $s = \frac{du}{d\theta}$, and performing integration by parts in the term $\lambda^T \frac{d}{dt}\frac{du}{d\theta}$ we derive

$$\begin{aligned} \frac{dL}{d\theta} = \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta} \right) dt - \int_{t_0}^{t_1} \left(-\frac{d\lambda^T}{dt} - \lambda^T \frac{\partial f}{\partial u} - \frac{\partial h}{\partial u} \right) s(t)\, dt \\ - \Big(\lambda(t_1)^T s(t_1) - \lambda(t_0)^T s(t_0) \Big). \end{aligned} \tag{68}$$

Now, we can force some of the terms in the last equation to be zero by solving the following adjoint differential equation for $\lambda(t)^T$ in backwards mode

$$\frac{d\lambda}{d\theta} = -\left(\frac{\partial f}{\partial u} \right)^T \lambda - \left(\frac{\partial h}{\partial u} \right)^T, \tag{69}$$

with final condition $\lambda(t_1) = 0$.

It is easy to see that this derivation is equivalent to solving the Karush-Kuhn-Tucker (KKT) conditions.

B Supplementaty code

References

1. Ascher, U. M. *Numerical methods for evolutionary differential equations*. SIAM, 2008.
2. Ascher, U. M. and Greif, C. *A First Course in Numerical Methods*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/9780898719987.
3. Barton, R. R. "Computing Forward Difference Derivatives In Engineering Optimization". In: *Engineering Optimization* 20.3 (1992), pp. 205–224. ISSN: 0305-215X. DOI: 10.1080/03052159208941281.
4. Bauer, F. L. "Computational Graphs and Rounding Error". In: *SIAM Journal on Numerical Analysis* 11.1 (1974), pp. 87–96. ISSN: 0036-1429. DOI: 10.1137/0711010.
5. Baydin, A. G. et al. "Automatic Differentiation in Machine Learning: A Survey". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 5595–5637. ISSN: 1532-4435.
6. Bennett, C. H. "Logical Reversibility of Computation". In: *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532. ISSN: 0018-8646. DOI: 10.1147/rd.176.0525.
7. Betancourt, M. "A Conceptual Introduction to Hamiltonian Monte Carlo". In: *arXiv* (2017). DOI: 10.48550/arxiv.1701.02434.
8. Bezanson, J. et al. "Julia: A Fast Dynamic Language for Technical Computing". In: *arXiv* (2012). DOI: 10.48550/arxiv.1209.5145.
9. Bezanson, J. et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. ISSN: 0036-1445. DOI: 10.1137/141000671.
10. Bolibar, J. et al. "Universal Differential Equations for glacier ice flow modelling". In: *Geoscientific Model Development* 16.22 (2023), pp. 6671–6687. DOI: 10.5194/gmd-16-6671-2023.
11. Bradley, A. M. *PDE-constrained optimization and the adjoint method*. Tech. rep. Technical Report. Stanford University. https://cs. stanford. edu/ textasciitilde ambrad~..., 2013.
12. Bui-Thanh, T. et al. "Extreme-scale UQ for Bayesian inverse problems governed by PDEs". In: *IEEE Computer Society Press* (2012), p. 3. URL: https://dl.acm.org/citation.cfm?id=2389000.
13. Chen, R. T. et al. "Neural ordinary differential equations". In: *Advances in neural information processing systems* 31 (2018).
14. Clifford. "Preliminary sketch of biquaternions". In: *Proceedings of the London Mathematical Society* 1.1 (1871), pp. 381–395.
15. Coveney, P. V., Dougherty, E. R., and Highfield, R. R. "Big data need big theory too". In: *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences* 374.2080 (2016), pp. 20160153–11. ISSN: 1364-503X. DOI: 10.1098/rsta.2016.0153. URL: http://rsta.royalsocietypublishing.org/lookup/doi/10.1098/rsta.2016.0153.

16. Cox, D. R. and Efron, B. "Statistical thinking for 21st century scientists". In: *Science Advances* 3.6 (2017), e1700768. DOI: 10.1126/sciadv.1700768. URL: http://advances.sciencemag.org/lookup/doi/10.1126/sciadv.1700768.

17. Cuffey, K. and Paterson, W. *The Physics of Glaciers*. Elsevier Science, 2010. ISBN: 978-0-08-091912-6. URL: https://books.google.fr/books?id=Jca2v1u1EKEC.

18. Dandekar, R., Rackauckas, C., and Barbastathis, G. "A Machine Learning-Aided Global Diagnostic and Comparative Tool to Assess Effect of Quarantine Control in COVID-19 Spread". In: *Patterns* 1.9 (2020), p. 100145. ISSN: 2666-3899. DOI: 10.1016/j.patter.2020.100145.

19. Elliott, C. "The simple essence of automatic differentiation". In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), p. 70. DOI: 10.1145/3236765.

20. Elliott, J. and Peraire, J. "Aerodynamic design using unstructured meshes". In: *Fluid Dynamics Conference* (1996). This has an example of the hardcore adjoint method implemented for aerodynamics. It may help to read this to see how the adjoint equations is being solved and the size of the problem. DOI: 10.2514/6.1996-1941.

21. Farrell, P. E. et al. "Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs". In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C369–C393. DOI: 10.1137/120873558. eprint: https://doi.org/10.1137/120873558. URL: https://doi.org/10.1137/120873558.

22. Gelbrecht, M. et al. "Differentiable programming for Earth system modeling". In: *Geoscientific Model Development* 16.11 (2023), pp. 3123–3135. DOI: 10.5194/gmd-16-3123-2023. URL: https://gmd.copernicus.org/articles/16/3123/2023/.

23. Gelman, A. et al. *Bayesian data analysis*. CRC press, 2013.

24. Ghattas, O. and Willcox, K. "Learning physics-based models from data: perspectives from inverse problems and model reduction". In: *Acta Numerica* 30 (2021), pp. 445–554. ISSN: 0962-4929. DOI: 10.1017/s0962492921000064.

25. Giering, R. and Kaminski, T. "Recipes for adjoint code construction". In: *ACM Trans Math Softw* 24.4 (1998), pp. 437–474. ISSN: 0098-3500. DOI: 10.1145/293686.293695. URL: https://doi.org/10.1145/293686.293695.

26. Giering, R. and Kaminski, T. "Recipes for adjoint code construction". In: *ACM Transactions on Mathematical Software (TOMS)* 24.4 (1998), pp. 437–474. ISSN: 0098-3500. DOI: 10.1145/293686.293695.

27. Giles, M. B. and Pierce, N. A. "An Introduction to the Adjoint Approach to Design". In: *Flow, Turbulence and Combustion* 65.3–4 (2000), pp. 393–415. ISSN: 1386-6184. DOI: 10.1023/a:1011430410075.

28. Goldberg, D. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163.

29. Gorban, A. and Wunsch, D. "The general approximation theorem". In: *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227)*. Vol. 2. 1998, 1271–1274 vol.2. DOI: 10.1109/IJCNN.1998.685957.

30. Griewank, A. "On Automatic Differentiation". In: (Feb. 1997).

31. Griewank, A. "Who invented the reverse mode of differentiation". In: *Documenta Mathematica, Extra Volume ISMP* 389400 (2012).

32. Griewank, A. and Walther, A. *Evaluating Derivatives*. 2008. ISBN: 978-0-89871-659-7. DOI: 10.1137/1.9780898717761. URL: https://doi.org/10.1137/1.9780898717761.

33. Griewank, A. and Walther, A. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

34. Hager, W. W. "Runge-Kutta methods in optimal control and the transformed adjoint system". In: *Numerische Mathematik* 87.2 (2000), pp. 247–282. ISSN: 0029-599X. DOI: 10.1007/s002110000178.

35. Hindmarsh, A. C. et al. "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 363–396.

36. Innes, M. "Don't Unroll Adjoint: Differentiating SSA-Form Programs". In: *arXiv* (2018).

37. Innes, M. et al. "A Differentiable Programming System to Bridge Machine Learning and Scientific Computing". In: *arXiv* (2019). DOI: 10.48550/arxiv.1907.07587.

38. Ipsen, I. C. F. and Meyer, C. D. "The Idea Behind Krylov Methods". In: *The American Mathematical Monthly* 105.10 (1998), pp. 889–899. ISSN: 0002-9890. DOI: 10.1080/00029890.1998.12004985.

39. Jameson, A. "Aerodynamic design via control theory". In: *Journal of Scientific Computing* 3.3 (1988), pp. 233–260. ISSN: 0885-7474. DOI: 10.1007/bf01061285.

40. Jensen, J. S., Nakshatrala, P. B., and Tortorelli, D. A. "On the consistency of adjoint sensitivity analysis for structural optimization of linear dynamic problems". In: *Structural and Multidisciplinary Optimization* 49.5 (2014), pp. 831–837. ISSN: 1615-147X. DOI: 10.1007/s00158-013-1024-4.

41. Johnson, S. G. "Notes on Adjoint Methods for 18.335". In: 2012.

42. Jouvet, G. "Inversion of a Stokes glacier flow model emulated by deep learning". In: *Journal of Glaciology* (2022), pp. 1–14. ISSN: 0022-1430. DOI: 10.1017/jog.2022.41.

43. Jouvet, G. et al. "Deep learning speeds up ice flow modelling by several orders of magnitude". In: *Journal of Glaciology* (2021), pp. 1–14. ISSN: 0022-1430. DOI: 10.1017/jog.2021.120.

44. Kantorovich, L. V. "On a mathematical symbolism convenient for performing machine calculations". In: *Dokl. Akad. Nauk SSSR*. Vol. 113. 4. 1957, pp. 738–741.

45. Karczmarczuk, J. “Functional Differentiation of Computer Programs”. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP ’98. Baltimore, Maryland, USA: Association for Computing Machinery, 1998, pp. 195–203. isbn: 1581130244. doi: 10.1145/289423.289442. url: https://doi.org/10.1145/289423.289442.

46. Kidger, P., Chen, R. T. Q., and Lyons, T. J. “"Hey, that’s not an ODE": Faster ODE Adjoints via Seminorms”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by M. Meila and T. Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 5443–5452. url: https://proceedings.mlr.press/v139/kidger21a.html.

47. Kim, S. et al. “Stiff neural ordinary differential equations”. en. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 31.9 (Sept. 2021), p. 093122. issn: 1054-1500, 1089-7682. doi: 10.1063/5.0060697. url: https://aip.scitation.org/doi/10.1063/5.0060697 (visited on 02/25/2022).

48. Lantoine, G., Russell, R. P., and Dargent, T. “Using Multicomplex Variables for Automatic Computation of High-Order Derivatives”. In: *ACM Transactions on Mathematical Software (TOMS)* 38.3 (2012), p. 16. issn: 0098-3500. doi: 10.1145/2168773.2168774.

49. Laue, S. *On the Equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation*. 2019. doi: 10.48550/ARXIV.1904.02990. url: https://arxiv.org/abs/1904.02990.

50. Leis, J. R. and Kramer, M. A. “Algorithm 658: ODESSA–an Ordinary Differential Equation Solver with Explicit Simultaneous Sensitivity Analysis”. In: *ACM Trans. Math. Softw.* 14.1 (Mar. 1988), pp. 61–67. issn: 0098-3500. doi: 10.1145/42288.214371. url: https://doi.org/10.1145/42288.214371.

51. Li, X. et al. “Scalable gradients for stochastic differential equations”. In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2020, pp. 3870–3882.

52. Lyness, J. N. “Numerical algorithms based on the theory of complex variable”. In: *Proceedings of the 1967 22nd national conference on* - (1967), pp. 125–133. doi: 10.1145/800196.805983.

53. Lyness, J. N. and Moler, C. B. “Numerical Differentiation of Analytic Functions”. In: *SIAM Journal on Numerical Analysis* 4.2 (1967), pp. 202–210. issn: 0036-1429. doi: 10.1137/0704019.

54. Ma, Y. et al. “A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions”. In: *arXiv:1812.01892 [cs]* (July 2021). arXiv: 1812.01892. url: http://arxiv.org/abs/1812.01892 (visited on 02/25/2022).

55. Manzyuk, O. et al. “Perturbation confusion in forward automatic differentiation of higher-order functions”. In: *Journal of Functional Programming* 29 (2019), e12.

56. Martins, J. R. R. A., Sturdza, P., and Alonso, J. J. "The complex-step derivative approximation". In: *ACM Transactions on Mathematical Software (TOMS)* 29 (2003), pp. 245–262. ISSN: 0098-3500. DOI: 10.1145/838250.838251.

57. Mathur, R. "An analytical approach to computing step sizes for finite-difference derivatives". PhD thesis. 2012.

58. Mitusch, S. K., Funke, S. W., and Dokken, J. S. "dolfin-adjoint 2018.1: automated adjoints for FEniCS and Firedrake". In: *Journal of Open Source Software* 4.38 (2019), p. 1292. DOI: 10.21105/joss.01292. URL: https://doi.org/10.21105/joss.01292.

59. Molnar, C., Casalicchio, G., and Bischl, B. "Interpretable Machine Learning – A Brief History, State-of-the-Art and Challenges". In: *arXiv* (2020). DOI: 10.48550/arxiv.2010.09337. eprint: 2010.09337.

60. Moses, W. and Churavy, V. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 12472–12485. URL: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf.

61. Naumann, U. *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9781611972078. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972078. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972078.

62. Neuenhofen, M. "Review of theory and implementation of hyper-dual numbers for first and second order automatic differentiation". In: *arXiv* (2018). DOI: 10.48550/arxiv.1801.03614.

63. Norcliffe, A. and Deisenroth, M. P. "Faster Training of Neural ODEs Using Gauß-Legendre Quadrature". In: *arXiv* (2023). DOI: 10.48550/arxiv.2308.10644.

64. Oden, J. T., Moser, R., and Ghattas, O. "Computer Predictions with Quantified Uncertainty, Part II". In: *SIAM News* 43.10 (2010), pp. 1–4.

65. Oktay, D. et al. "Randomized Automatic Differentiation". In: *arXiv* (2020). DOI: 10.48550/arxiv.2007.10412.

66. Onken, D. and Ruthotto, L. "Discretize-Optimize vs. Optimize-Discretize for Time-Series Regression and Continuous Normalizing Flows". In: *arXiv* (2020). DOI: 10.48550/arxiv.2005.13420.

67. Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN: 0201055945.

68. Rackauckas, C. et al. "Universal differential equations for scientific machine learning". In: *arXiv preprint arXiv:2001.04385* (2020).

69. Raissi, M., Perdikaris, P., and Karniadakis, G. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.10.045.

70. Ramsay, J. and Hooker, G. *Dynamic data analysis*. Springer, 2017.

71. Ramsundar, B., Krishnamurthy, D., and Viswanathan, V. "Differentiable Physics: A Position Piece". In: *arXiv* (2021). DOI: 10.48550/arxiv.2109.07573.

72. Razavi, S. et al. "The Future of Sensitivity Analysis: An essential discipline for systems modeling and policy support". In: *Environmental Modelling & Software* 137 (2021), p. 104954. DOI: 10.1016/j.envsoft.2020.104954. URL: http://doi.org/10.1016/j.envsoft.2020.104954.

73. Revels, J., Lubin, M., and Papamarkou, T. "Forward-Mode Automatic Differentiation in Julia". In: *arXiv:1607.07892 [cs.MS]* (2016). URL: https://arxiv.org/abs/1607.07892.

74. Rüde, U. et al. "Research and Education in Computational Science and Engineering". In: *SIAM Review* 60.3 (2018), pp. 707–754. ISSN: 0036-1445. DOI: 10.1137/16m1096840. URL: https://epubs.siam.org/doi/10.1137/16M1096840.

75. Ruder, S. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

76. Rudin, C. et al. "Interpretable machine learning: Fundamental principles and 10 grand challenges". In: *Statistic Surveys* 16.none (2022), pp. 1–85. ISSN: 1935-7516. DOI: 10.1214/21-ss133.

77. Sandu, A. "On the properties of Runge-Kutta discrete adjoints". In: *Computational Science–ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV 6*. Springer. 2006, pp. 550–557.

78. Sandu, A. "Solution of inverse problems using discrete ODE adjoints". In: *Large-Scale Inverse Problems and Quantification of Uncertainty* (2011), pp. 345–365.

79. Schäfer, F. et al. "AbstractDifferentiation.jl: Backend-Agnostic Differentiable Programming in Julia". In: *arXiv* (2021). DOI: 10.48550/arxiv.2109.12449.

80. Schanen, M. et al. "Transparent Checkpointing for Automatic Differentiation of Program Loops Through Expression Transformations". In: (2023). Ed. by J. Mikyška et al., pp. 483–497.

81. Serban, R. and Hindmarsh, A. C. "CVODES: the sensitivity-enabled ODE solver in SUNDIALS". In: *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 47438. 2005, pp. 257–269.

82. Shen, C. et al. "Differentiable modelling to unify machine learning and physical models for geosciences". In: *Nature Reviews Earth & Environment* (2023), pp. 1–16. DOI: 10.1038/s43017-023-00450-9.

83. Sirkes, Z. and Tziperman, E. "Finite Difference of Adjoint or Adjoint of Finite Difference?" In: *Monthly Weather Review* 125.12 (1997), pp. 3373–3378. ISSN: 0027-0644. DOI: 10.1175/1520-0493(1997)125<3373:fdoaoa>2.0.co;2.

84. Siskind, J. M. and Pearlmutter, B. A. "Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD". In: (2005).

85. Squire, W. and Trapp, G. "Using Complex Variables to Estimate Derivatives of Real Functions". In: 40 (1998), pp. 110–112. ISSN: 0036-1445. DOI: 10.1137/s003614459631241x.

86. Stoer, J. and Bulirsch, R. *Introduction to numerical analysis*. Springer, 2002.

87. Tsitouras, C. "Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption". In: *Computers & Mathematics with Applications* 62.2 (2011), pp. 770–775. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2011.06.002.

88. Vallis, G. K. "Geophysical fluid dynamics: whence, whither and why?" In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 472.2192 (2016), pp. 20160140–23. DOI: 10.1098/rspa.2016.0140. URL: http://rspa.royalsocietypublishing.org/lookup/doi/10.1098/rspa.2016.0140.

89. Wang, F. et al. "Backpropagation with Continuation Callbacks:Foundations for Efficient and ExpressiveDifferentiable Programming". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), p. 96. DOI: 10.1145/3341700.

90. Wang, Q., Hu, R., and Blonigan, P. "Least Squares Shadowing sensitivity analysis of chaotic limit cycle oscillations". In: *Journal of Computational Physics* 267 (2014), pp. 210–224. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2014.03.002. URL: https://www.sciencedirect.com/science/article/pii/S0021999114001715.

91. Wang, Y., Lai, C.-Y., and Cowen-Breen, C. "Discovering the rheology of Antarctic Ice Shelves via physics-informed deep learning". In: (2022).

92. Wengert, R. E. "A simple automatic derivative evaluation program". In: *Communications of the ACM* 7.8 (1964), pp. 463–464. ISSN: 0001-0782. DOI: 10.1145/355586.364791.

93. Wigner, E. P. "The unreasonable effectiveness of mathematics in the natural sciences". In: *Communications on Pure and Applied Mathematics* 13 (1960), pp. 1–14. DOI: 10.1002/cpa.3160130102. URL: https://doi.org/10.1002/cpa.3160130102.

94. Wolfe, P. "Checking the Calculation of Gradients". In: *ACM Transactions on Mathematical Software (TOMS)* 8.4 (1982), pp. 337–343. ISSN: 0098-3500. DOI: 10.1145/356012.356013.

95. Zdeborová, L. "Understanding deep learning is also a job for physicists". en. In: *Nature Physics* (May 2020). ISSN: 1745-2473, 1745-2481. DOI: 10.1038/s41567-020-0929-2. URL: http://www.nature.com/articles/s41567-020-0929-2 (visited on 05/29/2020).

96. Zhang, H. and Sandu, A. "FATODE: A library for forward, adjoint, and tangent linear integration of ODEs". In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C504–C523.

97. Zhuang, J. et al. "Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE." In: *Proceedings of machine learning research* 119 (2020), pp. 11639–11649.