Differentiable Programming for Differential Equations: A Review

Facundo Sapienza*[1], Jordi Bolibar[2, 3], Frank Schäfer[4], Brian Groenke[5,6], Avik Pal[4], Victor Boussange[7], Patrick Heimbach[8,9], Giles Hooker[10], Fernando Pérez[1], Per-Olof Persson[11], and Christopher Rackauckas[12,13]

[1]*Department of Statistics, University of California, Berkeley (USA)*
[2]*TU Delft, Department of Geosciences and Civil Engineering, Delft (Netherlands)*
[3]*Univ. Grenoble Alpes, CNRS, IRD, G-INP, Institut des Géosciences de l'Environnement, Grenoble, France*
[4]*CSAIL, Massachusetts Institute of Technology, Cambridge (USA)*
[5]*TU Berlin, Department of Electrical and Computer Engineering, Berlin (Germany)*
[6]*Environmental Research Centre, Leipzig (Germany)*
[7]*Swiss Federal Research Institute WSL, Birmensdorf (Switzerland)*
[10]*Department of Statistics and Data Science, University of Pennsylvania (USA)*
[8]*Department of Mathematics, University of California, Berkeley (USA)*
[9]*Oden Institute for Computational Engineering and Sciences, University of Texas at Austin (USA)*
[11]*Jackson School of Geosciences, University of Texas at Austin (USA)*
[12]*Massachusetts Institute of Technology, Cambridge (USA)*
[13]*JuliaHub, Cambridge (USA)*

May 5, 2024

*Corresponding author: `fsapienza@berkeley.edu`

Abstract

The differentiable programming paradigm has become a central component of modern machine learning and scientific computing techniques. A long tradition of this paradigm exists in the context of inverse methods, involving differential equation-constrained, gradient-based optimization. The recognition of strong parallels 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 the associated gradient loss requires to differentiate the numerical solutions to the differential equation. This task is non-trivial and requires careful algebraic and numeric manipulations of the underlying dynamical system. Here, we provide a comprehensive review of existing techniques to compute derivatives of numerical solutions of differential equation systems. We first discuss the importance of gradients of solutions of ODEs in a variety of scientific domains. Second, we lay out the mathematical foundations of the various approaches and compare them with each other. Finally, we delve into the computational considerations and explore the solutions available in modern scientific software. By delivering an exhaustive review of sensitivity methods, we hope that this work accelerates the fusion of scientific models and data, and fosters a modern approach to scientific modelling.

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

Contents

Plain language summary

Differential equations are mathematical objects used as scientific models to explicitly describe the processes and dynamics within various systems, based on prior domain knowledge. They are fundamental in many scientific disciplines for modeling phenomena such as physical processes, population dynamics, social interactions, and chemical reactions. By contrast, data-driven models do not necessarily require a detailed understanding of the underlying physical processes, and learn patterns and relationships directly from data. By embedding assumptions on the system dynamics in their structure, traditional mathematical models predictions are constrained, and as such require few data to reproduce a phenomenon. Yet, their structure make them also less amenable to assimilate data, and they may fail at describing complex processes. On the other hand, data-driven models are particularly useful when the underlying processes are poorly understood or complex, but require a significantly larger amount of data. The combination of mechanistic models with data-driven models is becoming increasingly common in many scientific domains. To achieve this, these models need to leverage both domain knowledge and data, 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. In this review, we present 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

Evaluating how the value of a function changes with respect to its arguments and parameters, i.e., its gradient, plays a central role in optimization, sensitivity analysis, Bayesian inference, inverse methods, and uncertainty quantification, among many (Razavi et al. 2021). Modern machine learning methods require the use of gradients to efficiently navigate within the high-dimensional space defined by its parameters (e.g., the weights of a neural network), in order to find the set of parameter values for which the model behavior best matches data. When optimizing an objective function, gradient-based methods (for example, gradient descent and its many variants (Ruder 2016)) are more efficient at finding a minimum and converge faster to them than gradient-free methods. When numerically computing the posterior of a probabilistic model, gradient-based sampling strategies are better at estimating the posterior distribution than gradient-free methods (Neal et al. 2011). The availability of the Hessian further helps to improve the convergence rates of these algorithms (Bui-Thanh et al. 2012). Furthermore, the *curse of dimensionality* renders gradient-free optimization and sampling methods computationally intractable for most large-scale problems.

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.

Models based on differential equations arising in simulation-based science, which play a central role in describing the behaviour of systems in natural and social sciences, are not an exception to the rule (Ghattas et al. 2021). The solution of differential equations can be seen as functions that map parameter and initial conditions to state variables, similar to machine learning models. Some authors have recently suggested differentiable programming as the bridge between modern machine learning and traditional scientific models (Gelbrecht et al. 2023; Rackauckas et al. 2021; Ramsundar et al. 2021; Shen et al. 2023). Being able to compute gradients or sensitivities of dynamical systems opens the door to more complex data assimilation models that leverage strong physical priors while offering flexibility to adapt to observations. This is very appealing in fields such as computational physics, geophysics, and biology, to mention a few, 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.

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

Sensitivity analysis refers 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) input 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 (Ramsay et al. 2017). 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 (Giles et al. 2000b), 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 (Raissi et al. 2019). On the other hand, there has been an increased interest in augmenting differential equation model with data-driven components, such as universal differential equations (Dandekar et al. 2020; Rackauckas et al. 2020), which also includes the case of neural ordinary differential equations (Chen et al. 2018) and neural stochastic differential equations (Li et al. 2020b).

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 approximate their solution. This means that solutions cannot be directly differentiated and require special treatment to compute derivatives. Furthermore, numerical solutions introduce approximation errors. These errors can be propagated to the computation of the gradient, leading to incorrect gradient values. Alternatively, there is a broad literature on numerical methods for solving differential equations (Hairer et al. 2008; Wanner et al. 1996). 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. As coined by Uwe Naumann, *the automatic generation of optimal (in terms of robustness and efficiency) adjoint versions of large-scale simulation code is one of the great open challenges in the field of High-Performance Scientific Computing* (Naumann 2011). The second goal of this article is to review different methods that exist to achieve this goal.

> **Question 2.** *How can one efficiently 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 to compute derivatives by sequentially applying the chain rule to the sequence of unit operations that constitute a computer program (Griewank et al. 2008; Naumann 2011). The premise is simple: every computer program is ultimately an algorithm described by a nested concatenation of elementary algebraic operations, such as addition and multiplication, that are individually easy to differentiate and their composition is easy to differentiate by using the chain rule (Giering et al. 1998a). More broadly than AD, differentiable programming encapsulates the set of software tools that allows to compute efficient and robust gradients though complex algorithms, including numerical solvers (Innes et al. 2019). 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 and/or compute an intermediate adjoint. Furthermore, it is important to be aware that when using AD or any other technique we are differentiating the algorithm used to compute the numerical solution, not the numerical solution itself, which can lead to wrong results (Eberhard et al. 1996).

The differences between methods to compute sensitivities 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 total number of parameters and size of the differential equation system, 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?*

Differentiable programming is opening new ways of doing research across different domains of science and engineering. Arguably, its potential has so far been under-explored but is being rediscovered in the age of data-driven science. Realizing its full potential, requires collaboration between domain scientists, methodological scientists, computational scientists, and computer scientists in order to develop successful, scalable, practical, and efficient frameworks for real world applications. As we make progress in the use of these tools, new methodological questions emerge. How do these methods compare? How can they be improved? In this review we present a comprehensive list of methods that exists at the intersection of differentiable programming and differential equation modelling.

This review is structured in three main sections, looking at differentiable programming for differential equations from three different perspectives: a domain science perspective (Section 2), a mathematical perspective (Section 3) and a computer science perspective (Section 4).

2 Scientific motivation: A domain science perspective

Mechanistic (or process-based) models have played a central role in a wide range of scientific disciplines. They consist of precise mathematical descriptions of physical mechanisms that include the modelling of causal interactions, feedback loops and dependencies between components of the system under consideration (Rackauckas et al. 2020). These mathematical representations typically take the form of differential equations. Together with the numerical methods to approximate their solutions, differential equations have led to fundamental advances in the understanding and prediction of physical and biological systems. They depend on uncertain inputs or parameters that determine the processes represented. The parameter values have traditionally been estimated independently of the model, which poses several problems (Hartig et al. 2012). First, the independent estimation of parameters and processes rapidly becomes impossible as the number of state variables increases, especially when considering highly non-linear processes. Second, the measurement of certain parameters – to the extent that they can be measured – are intrinsically difficult (Schartau et al. 2017). Third, parameter values estimated from laboratory experiments, or those representing subgrid-scale processes, may be specific to the experimental setting, or to the resolution of a simulation, and resulting simulations are likely to diverge from observations in the field.

Due to the difficulty of estimating parameter values in mechanistic models, and accompanying the massive growth of data upon which they depend, statistical (or machine learning) models have led the modelling field in the past decades (Cox et al. 2017). Advances in the field of machine learning, and particularly in deep learning (LeCun et al. 2015), allowed statistical models to learn at multiple levels of abstraction and capture extremely complex nonlinear patterns and information hidden in large datasets. However, whereas the use of machine learning models for regression, i.e., interpolation, has shown great successes, its use for prediction, i.e., extrapolation, has been heavily criticized, as they critically assume that patterns contained in observed data will repeat in the future, which may not be the case (Barnosky et al. 2012; Dormann 2007). In contrast to purely statistical models, the process knowledge embedded in the structure of mechanistic models renders them more robust for predicting dynamics under different conditions.

The fields of mechanistic modelling and statistical modelling have mostly evolved independently (Zdeborová 2020), due to several reasons. On the one hand, domain scientists have often been reluctant in learning about machine learning methods, judging them as opaque black boxes, unreliable, and not respecting domain-established knowledge (Coveney et al. 2016). 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 (Abarbanel et al. 2018; Bocquet et al. 2019; Brajard et al. 2021; Carrassi et al. 2018; Curtsdotter et al. 2019; Gábor et al. 2015; Gharamti et al. 2017; Molnar et al. 2020; Rasp et al. 2018; Rosenbaum et al. 2019; Rudin et al. 2022; Schneider et al. 2017; Toms et al. 2020; Yazdani et al. 2020). Inverse modelling is an attempt to bridge the statistical and mechanistic modelling fields (Rüde et al. 2018; Wigner 1960). Differentiable programming is key in this process.

2.1 Domain-specific applications

Arguably, the notion of differentiable programming has a long tradition in computational physics which is founded on solving and/or inverting models based on differential equations. The overarching goal of inverse modelling is to find a set of optimal model parameters that minimizes an objective or cost function quantifying the misfit between observations and the simulated state. Depending on the scientific problem, inverse modelling can be performed on different quantities of interest of system under study, including:

- **Initial conditions.** Inverting for uncertain initial conditions, which, when integrated using the model, leads to an optimal match between the observations and the simulated state (or diagnostics); variants thereof are used for optimal forecasting.

- **Boundary conditions.** Inverting for uncertain surface (e.g., interface fluxes), bottom (e.g., bed properties), or lateral (e.g., open boundaries of a limited domain) boundaries, which, when used in the model, produces 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. Parametrization is a special case of parameter inversion, where a parametric function (e.g., a neural network) is used to approximate processes.

Besides the use of sensitivity methods 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, non-normal amplification of specific norms of interest; and characterizing and quantifying uncertainties by way of second derivative (Hessian) information.

In recent years the use of machine learning methods has become more popular in many scientific domains (**ml_clouds_climate**; **chem_ml_review**; **bio_ml_review**). 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 (Gorban et al. 1998). This combined flexibility can be exploited by many different domain-specific problems to tailor modelling needs to both dynamics and data characteristics.

In the following, we present selected examples belonging to a wide range of scientific communities where differentiable programming techniques have been used.

2.1.1 Computational physics and optimal design

There is a long tradition of computational physics models based on adjoint methods and automatic differentiation pipelines. These include examples in fluid dynamics, where sensitivity methods have been used for optimal design and optimal control (Lions 1971; Pironneau 2005), aerodynamic design and shape optimization (Allaire et al. 2014; Giles et al. 2000b; Jameson 1988; Mohammadi et al. 2009) and supersonic aircraft design (Fike 2013; Hu 2010). Furthermore, these methods have been applied to particle physics (Dorigo et al. 2022) and quantum chemistry (Arrazola et al. 2021), to the optimal design of nanophotonics devices (Molesky et al. 2018), to optimal design in electromagnetism (Georgieva et al. 2002), to stellarator coil design (McGreivy et al. 2021), and to biological applications (Strouwen et al. 2022).

2.1.1.1 Computational fluid dynamics

2.1.1.2 Quantum physics

Quantum optimal control has diverse applications spanning a broad spectrum of quantum systems. Optimal control methods have been used to optimize pulse sequences, enabling the design of high-fidelity quantum gates and the preparation of complex entangled quantum states. Typically, the objective is to maximize the fidelity to a target state or unitary operation, accompanied by additional constraints or costs specific to experimental demands. The predominant control algorithms are gradient-based optimization methods, such as gradient ascent pulse engineering (GRAPE), and rely on the computation of derivatives for solutions of the differential equations modeling the time evolution of the quantum system. In cases where the analytical calculation of a gradient is impractical, numerical evaluation using AD becomes a viable alternative (Abdelhafez et al. 2020, 2019; Goerz et al. 2022; Jirari 2009, 2019; Leung et al. 2017; Schäfer et al. 2020). Specifically, AD streamlines the adjustment to diverse objectives or constraints, and its efficiency can be enhanced by employing custom derivative rules for the time propagation of quantum states as governed by solutions to the Schrödinger equation (Goerz et al. 2022). Moreover, sensitivity methods for differential equations facilitate the design of feedback control schemes necessitating the differentiation of solutions to stochastic differential equations (Schäfer et al. 2021a).

2.1.2 Geosciences

Many geoscientific phenomena are governed by global and local conservation laws 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 (Vallis 2016), describing geophysical properties of many Earth system components, such as the atmosphere, ocean, land surface, and glaciers. In such models, calibrating model parameters is extremely challenging, due to datasets being sparse in both space and time, heterogeneous, and noisy; and computational models involving high-dimensional parameter spaces, often on the order of $O(10^3) - O(10^8)$. Moreover, many existing mechanistic models can only partially describe observations, with many detailed physical processes being ignored or poorly parameterized.

2.1.2.1 Meteorology

Numerical weather prediction (NWP) is among the most prominent fields where adjoint methods have played an important role (Errico 1997). Adjoint methods were introduced to infer initial conditions that minimize the misfit between simulations and weather observations (Courtier et al. 1987; Talagrand et al. 1987), with the value of second-derivative information also being recognized (Dimet et al. 2002). This led to the development of the so-called *four-dimensional variational* (4D-Var) data assimilation (DA) technique (Rabier et al. 2000; Rabier et al. 1992) at the European Centre for Medium-Range Weather Forecasts (ECMWF) as one the most advanced DA approaches, and which contributed substantially to the *quiet revolution* in NWP (Bauer et al. 2015). Related, within the framework of transient non-normal amplification or optimal excitation (Farrell 1988; Farrell et al. 1996), the adjoint method has been used extensively to infer patterns in initial conditions that over time contribute to maximum uncertainty growth in forecasts (Buizza et al. 1995; Palmer et al. 1994) and to infer the so-called *Forecast Sensitivity-based Observation Impact* (FSOI) (Langland et al. 2004). Except in very few instances and for experimental purposes (Giering et al. 2006), automatic differentiation has not been used in the development of adjoint models in NWP. Instead, the adjoint code was derived and implemented manually.

2.1.2.2 Oceanography

The recognition of the benefit of adjoint methods for use in data assimilation in the ocean coincided roughly with that in meteorology (Thacker 1988; Thacker et al. 1988). The first application appeared soon thereafter in the context of a basin-scale general circulation model (Tziperman et al. 1992a,b; Tziperman et al. 1989). An important detail is that their work already differed from the "4D-Var" problem of NWP in that sensitivities were computed not only with respect to initial conditions but also with respect to surface boundary conditions, i.e., air-sea fluxes of buoyancy and momentum. Again, the value of the second-derivative for uncertainty quantification was readily realized (Thacker 1989). Similar to the work on calculating singular vectors in the atmosphere based on tangent linear and adjoint versions of a GCM to solve a generalized eigenvalue problem, the question of El Niño predictability invited model-based singular vector computations in models of the Tropical Pacific Ocean (Moore et al. 1997a,b). Such model-based singular vectors were also later computed for optimal excitations of the North Atlantic thermohalince circulation (Zanna et al. 2012; Zanna et al. 2011, 2010). Notably in the context of this review, the consortium for Estimating the Circulation and Climate of the Ocean (ECCO) (Stammer et al. 2002) set out in around 1999 to develop a parameter and state estimation framework, whereby a state-of-the-art ocean general circulation model is fit to diverse observations by way of PDE-constrained, gradient-based optimization, with the adjoint model of the GCM computing the gradient. Importantly, the adjoint model of the MIT general circulation model (MITgcm) is generated using source-to-source automatic differentiation (Heimbach et al. 2005; Marotzke et al. 1999), initially using the *Tangent linear and Adjoint Model Compiler* (TAMC)(Giering et al. 1998a) and then its commercial successor *Transformation of Algorithms in Fortran* (TAF) (Giering et al. 2006). Rigorous exploitation of AD enabled the simulation framework to be significantly extended over time in terms of vastly improved model numerics (Forget et al. 2015) and coupling other Earth system components, including biogeochemistry (Dutkiewicz et al. 2006), sea-ice (Heimbach et al. 2010), and sub-ice shelf cavities (Heimbach et al. 2012). Unlike NWP-type 4D-Var, the use of AD also enabled extension of the framework to the problem of parameter calibration from observations (Ferreira et al. 2005; Liu et al. 2012; Stammer 2005). Arguably, this work heralded much of today's efforts in online learning of parameterization schemes, where the functional representation between the parameters and the learning data are provided by the numerical implementation of a PDF rather than by a neural network. The desire to make AD for Earth system models written in Fortran (to date the vast majority) has also spurned the development of alternative AD tools with powerful reverse modes, notably OpenAD (Utke et al. 2008) and most recently Tapenade (Gaikwad et al. 2023, 2024; Hascoet et al. 2013). There is enormous potential to seamlessly integrate the inverse-modeling and machine-learning based approaches through the concept of differentiable programming.

2.1.2.3 Climate science

The same goals that have driven the use of sensitivity information in numerical weather prediction (optimal initial conditions for forecasts) or ocean science (state and parameter estimation) apply in the world of climate modeling. The recognition that good initial conditions (e.g., such that are closest to the real or observed system) will lead to improved forecasts on subseasonal, seasonal, interannual, or even decadal time scales has driven major community efforts (Meehl et al. 2021). However, there has been a lack so far in exploiting the use of gradient information to achieve optimal initialization for coupled Earth system models (Frolov et al. 2023). One conceptual challenge is the presence of multiple timescales in the coupled system and the utility of gradient information beyond many synoptic time scales in the atmosphere and ocean (Lea et al. 2000a, 2002). Nevertheless, efforts are underway to enable adjoint-based parameter estimation of coupled atmosphere-ocean climate models, with AD again playing a crucial role in generating the corresponding adjoint model (Blessing et al. 2014; Lyu et al. 2018; Stammer et al. 2018). Complementary, recognizing the power of differentiable programming, efforts are also targeting the development of *neural atmospheric general circulation models* in `JAX`, which combine a differentiable dynamical core with neural operators as surrogate models of unresolved physics (Kochkov et al. 2023).

2.1.2.4 Glaciology

Due to the difficulty of having direct observations of internal and basal rheological processes of glaciers, adjoint methods have been widely used to study them, following the pioneering work by (MacAyeal 1992) three decades ago. Since then, the adjoint method has been applied to many different studies investigating parameter and state estimation (Goldberg et al. 2013), ice volume sensitivity to basal, surface and initial conditions (Heimbach et al. 2009), inversion of initial conditions (Mosbeux et al. 2016) or inversion of basal friction (Morlighem et al. 2013). These studies either derived the adjoint with a manual implementation or combined AD with hand-written adjoint solvers. Additionally, the use of AD has become increasingly widespread in glaciology, paving the way for more complex modelling frameworks (Gaikwad et al. 2023; Hascoët et al. 2018). Recently, differentiable programming has also facilitated the development of hybrid frameworks, combining numerical methods with data-driven models by means of universal differential equations (Bolibar et al. 2023). Alternatively, some other approaches have dropped the use of numerical solvers in favour of different flavours of physics-informed neural networks, exploring the inversion of rheological properties of glaciers (Wang et al. 2022) and to accelerate ice thickness inversions and simulations by leveraging GPUs (Jouvet 2023; Jouvet et al. 2021).

2.1.3 Biology and ecology

Differential equation models have been broadly used in biology and ecology to model the dynamics of genes and alleles (PAGE et al. 2002), the ecological and evolutionary dynamics of biological units from bacteria to ecological communities (Åkesson et al. 2021; Boussange et al. 2023, 2022; Chalmandrier et al. 2021; Gábor et al. 2015; Lion 2018; van den Berg et al. 2022; Villa et al. 2021), and biomass and energy fluxes and transformation at ecosystem levels (Franklin et al. 2020; Geary et al. 2020; Schartau et al. 2017; Weng et al. 2015). Estimating parameters of biological models from laboratory experiments is costly and difficult(Schartau et al. 2017), and may even result in simulations failing in capturing real biological dynamics (Watts 2001). As a consequence, statistical models have been the main modelling paradigm in biology (Zimmermann et al. 2010), but inverse modelling methods are increasingly advocated to inform mechanistic models (Alsos et al. 2023; Hartig et al. 2012; Pantel et al. 2023). Inverse modelling can take the form of parameter estimation (Schartau et al. 2017) or model selection (Johnson et al. 2004). Provided that they are inferred together with uncertainties, parameters can be interpreted to better understand the strengths and effects of the processes considered (Curtsdotter et al. 2019; Godwin et al. 2020; Higgins et al. 2010; Pontarp et al. 2019). In model selection, candidate models embedding competing hypotheses about causal processes are derived, and the relative support of each model given the data is computed to discriminate between the hypotheses (Alsos et al. 2023; Johnson et al. 2004). The computation of the most probable model parameter values, or the computation of the different model supports, critically involves sensitivity methods which must adequately handle the typically larger number of parameters and the nonlinearities of biological models (Gábor et al. 2015).

While inverse modelling in biology has been mostly agnostic to differential programming techniques, their potential have recently been highlighted (Alsos et al. 2023; Frank 2022) and new approaches to accommodate for the specificity of biological and ecological models are increasingly proposed (Boussange et al. 2024; Paredes et al. 2023; Yazdani et al. 2020). Given that key ecological processes are not accurately represented in biological models (Chalmandrier et al. 2021; Hartig et al. 2012; Schartau et al. 2017), hybrid approaches where neural networks are used as data-driven parametrization of uncertain processes in differential equation models are particularly relevant (Boussange et al. 2024; Rasp et al. 2018) . Increasingly available biological dataset following the development of monitoring technologies such as paleao-time series (Alsos et al. 2023), environmental DNA (Ruppert et al. 2019), remote sensing (Jetz et al. 2019), bioaccoustics (Aide et al. 2013), and citizen observations (GBIF: The Global Biodiversity Information Facility 2022), are additional opportunities to leverage mechanistic models with data-driven components.

3 Methods: A mathematical perspective

There is a large family of methods for computing gradients or sensitivities of functions based on solutions of systems of differential equations. Depending on the number of parameters and the characteristics of the differential equation (e.g., level of stiffness), they have different mathematical, numerical, and computational advantages. These methods can be roughly classified as follows (Ma et al. 2021).

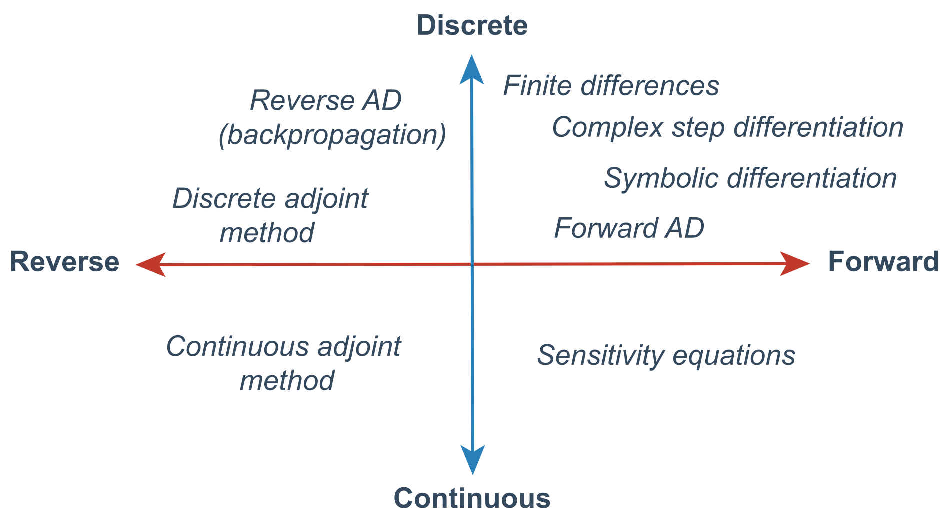

Figure 1: *Schematic representation of the different methods available for differentiation involving differential equation solutions. These can be classified depending if they find the gradient by solving a new system of differential equations (continuous) or if instead they manipulate unit algebraic 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 reverse methods.*

- *Continuous* vs *discrete* methods
- *Forward* vs *reverse* methods

Figure 1 displays a classification of some methods under this two-fold division.

The *continuous* vs *discrete* distinction is one of mathematical and numerical nature. When solving for the gradient of a function of the solution of a differential equation, one needs to derive both a mathematical expression for the gradient (the differentiation step) and solve the differential equations using a numerical solver (the discretization step) (Bradley 2013; Onken et al. 2020; Sirkes et al. 1997; Zhang et al. 2014). Depending on the order of these two operations, we refer to discrete methods (discretize-then-differentiate) or continuous methods (differentiate-then-discretize). In the case of *discrete* methods, gradients are computed based on simple function evaluations of the solutions of the numerical solver (finite differences, complex step differentiation) or by manipulation of atomic operations inside the numerical solver (AD, symbolic differentiation, discrete adjoint method). It is worth noting that although both approaches are subsumed under discrete methods, their numerical properties are quite different. In the case of *continuous* methods, a new set of differential equations is derived that allow the calculation of the desired gradient, namely the sensitivity (sensitivity equations) or the adjoint (continuous adjoint method) of the system. When comparing discrete to continuous methods, we are focusing, beyond computational efficiency, on the mathematical consistency of the method, that is, *is the method estimating the right gradient?*. Discrete methods compute the exact derivative of the numerical approximation to the loss function, but they do not necessarily yield to an approximation of the exact derivatives of the objective function ((Walther 2007), Section 3.9.4).

The distinction between *forward* and *reverse* methods relate to the timing of gradient computation, whether it occurs during the initial forward pass of the numerical solver or during a subsequent recalculation(Griewank et al. 2008). In all *forward* methods the solution of the differential equation is solved sequentially and simultaneously with the derivative (either the full gradient or more commonly a directional derivative) during the forward pass of the numerical solver. On the contrary, *reverse* methods compute the gradient tracking backwards the forward model by solving a new problem that moves in the opposite direction as the original numerical solver. For systems of ordinary differential equations (ODEs) and initial value problems (IVPs), most numerical methods solve the differential equation progressively moving forward in time, meaning that reverse methods then solve for the gradient moving backwards in time.

As discussed in the following sections, forward methods are very efficient for problems with a small number of parameters we want to differentiate with respect to, while reverse methods are more efficient for a large number of parameters but they come with a larger memory cost or compute overhead which needs to be overcome using different performance tricks. With the exception of finite differences and complex step differentiation, the rest of the forward methods (i.e. forward AD, sensitivity equations, symbolic differentiation) compute the full sensitivity of the differential equation, which can be computationally expensive or intractable for large systems. Conversely, reverse methods are based on the computation of intermediate variables, known as the adjoint or dual variables, that cleverly avoid the unnecessary calculation of the full sensitivity at expenses of larger memory cost (Givoli 2021).

The rest of this section is organized as follows. We first introduce some basic mathematical notions to facilitate the discussion of the sensitivity methods (Section 3.1). We then mathematically formalize each of the methods listed in Figure 1. We finally discuss the mathematical foundations of these methods in 3.9 with a comparison of some mathematical foundations of these methods.

3.1 Preliminaries

Consider a system of first-order ODEs given by

$$\frac{du}{dt} = f(u, \theta, t) \tag{1}$$

subject to the initial condition $u(t_0) = u_0$, where $u \in \mathbb{R}^n$ is the unknown solution vector of the ODEs, $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \mapsto \mathbb{R}^n$ is a function that depends on the state u, $\theta \in \mathbb{R}^p$ is a vector parameter, and t refers to time. Here, n denotes the total number of ODEs and p the number of parameters of the differential equation. Although we here 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 to the case, equation (1) also extends to partial differential equations (PDEs) (when discretized via e.g. the method of lines (Ascher 2008)), and to differential algebraic equations (DAEs) (Wanner et al. 1996). Except for a minority of functions $f(u, \theta, t)$, solutions to Equation (1) need to be computed using numerical solvers.

3.1.1 Numerical solvers for ordinary differential equations

Numerical solvers for the solution of ODEs or IVPs can be classified as one-step methods, among which Runge-Kutta methods are the most widely used, and multi-step methods (Hairer et al. 2008). Given an integer s, s-stage Runge-Kutta methods are defined by generalizing numerical integration quadrature rules as follows

$$\begin{aligned} u^{n+1} &= u^n + \Delta t_n \sum_{i=1}^{s} b_i k_i \\ k_i &= f\left(u^n + \sum_{j=1}^{s} a_{ij} k_j,\, \theta,\, t_n + c_i \Delta t_n \right) \qquad i = 1, 2, \dots, s. \end{aligned} \tag{2}$$

where $u^n \approx u(t_n)$ approximates the solution at time t_n, where $\Delta t_n = t_{n+1} - t_n$, and where a_{ij}, b_i, and c_j are scalar coefficients with $i, j = 1, 2, \dots, j$, usually represented in the form of a tableau. A Runge-Kutta method is called explicit if $a_{ij} = 0$ for $i \leq j$; diagonally implicit if $a_{ij} = 0$ for $i < j$; and fully implicit otherwise. Different choices of the number of stages s and coefficients give different orders of convergence of the numerical scheme (Butcher 2001; Butcher et al. 1996).

In contrast, multi-step linear solvers are of the form

$$\sum_{i=0}^{k_1} \alpha_{ni} u^{n-i} = \Delta t_n \sum_{j=0}^{k_2} \beta_{nj} f(u^{n-j}, \theta, t_{n-j}) \tag{3}$$

where α_{ni} and β_{nj} are numerical coefficients (Hairer et al. 2008). In most cases, including Adam method and backwards differentiation formulas (BDF), we have the coefficients $\alpha_{ni} = \alpha_i$ and $\beta_{nj} = \beta_j$, meaning that the coefficient do not depend on the iteration. Notice that multi-step linear methods are linear in the function f, which is not the case in Runge-Kutta methods with intermediate evaluations (Ascher 2008). Explicit methods are characterized by $\beta_{n,0} = 0$ and are easy to solve by direct iterative updates. For implicit methods, the usually non-linear equation

$$g_i(u_i; \theta) = u_i - h\beta_{n0} f(u_i, \theta, t_i) - \alpha_i = 0, \tag{4}$$

with α_i a computed coefficient that includes the information of all past iterations, can be solved using predictor-corrector methods (Hairer et al. 2008) or iteratively using Newton's method (Hindmarsh et al. 2005).

When choosing a numerical solver for differential equations, one crucial factor to consider is the stiffness of the equation. Stiffness encompasses various definitions, reflecting its historical development and different types of instabilities (Dahlquist 1985). Two definitions are noteworthy

- Stiff equations are equations for which explicit methods do not work and implicit methods work better (Wanner et al. 1996).
- Stiff differential equations are characterized by dynamics with different time scales (Kim et al. 2021), also characterized by the phenomena of increasing oscillations (Dahlquist 1985).

Stability properties can be achieved by different means, for example by the use of implicit methods or stabilized explicit methods, such as Runge–Kutta–Chebyshev (Der Houwen et al. 1980; Wanner et al. 1996). When using explicit methods, smaller timesteps may be required to guarantee stability.

Another important consideration is the choice of the time-steps Δt_i in a numerical solver (Hairer et al. 2008). Modern solvers include stepsize controllers that pick Δt_i as large as possible to minimize the total number of steps while preventing large errors in the numerical solution controlled by adjustable relative and absolute tolerances (see Appendix B).

3.1.2 What to differentiate and why?

In most applications, the need for differentiating the solution of ODEs stems from the need to obtaining the gradient of a function $L(u(\cdot, \theta))$ with respect to the parameter θ, where L can denote

- **A loss function or an empirical risk function**. This is usually a real-valued function that quantifies the level of agreement between the model prediction and observations. Examples of loss functions include the squared error

$$L(\theta) = \frac{1}{2} \| u(t_1; \theta) - u^{\text{target}}(t_1) \|_2^2, \tag{5}$$

where $u^{\text{target}}(t_1)$ is the desired target observation at some later time t_1, and $\| \cdot \|_2$ is the Euclidean norm. More generally, we can evaluate the loss function at points of the time series for which we have observations,

$$L(\theta) = \frac{1}{2} \sum_{i=1}^{N} \omega_i \, \| u(t_i; \theta) - u^{\text{target}}(t_i) \|_2^2. \tag{6}$$

with ω_i some arbitrary non-negative weights. More generally, misfit functions used in optimal estimation and control problems are composite maps from the parameter space θ via the model's state space, in this case, the solution $u(t)$, to the observation space defined by a new variable $y(t) = H(u(t, \theta))$, where $H : \mathbb{R}^n \mapsto \mathbb{R}^o$ is a given function mapping the latent state to observational space (Bryson et al. 1979). In these cases, the loss function generalizes to

$$L(\theta) = \frac{1}{2} \sum_{i=1}^{N} \omega_i \, \| H(u(t_i; \theta)) - y^{\text{target}}(t_i) \|_2^2. \tag{7}$$

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

$$L(u(\cdot, \theta)) = \int_{t_0}^{t_1} h(u(t; \theta), \theta) dt, \tag{8}$$

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 (Ramsay et al. 2017). Other methods like gradient matching and generalized smoothing the loss depends on smooth approximations of the trajectory and their derivatives.

- **The likelihood function or posterior probability.** From a statistical (and physical) perspective, it is common to assume that observations correspond to noisy observations of the underlying dynamical system, $y_i = H(u(t_i; \theta)) + \varepsilon_i$, with ε_i errors or residual that are independent of each other and of the trajectory $u(\cdot; \theta)$ (Ramsay et al. 2017). When H is the identity, each y_i corresponds to the noisy observation of the state $u(t_i; \theta)$. If $p(Y | t, \theta)$ is the probability distribution of $Y = (y_1, y_2, \dots, y_N)$, the maximum likelihood estimator (MLE) of θ is defined as

$$\theta^* = \underset{\theta}{\text{argmax}} \; \ell(Y | \theta) = \prod_{i=1}^{n} p(y_i | \theta, t_i). \tag{9}$$

When $\varepsilon_i \sim N(0, \sigma_i^2 \, \mathbb{I})$ is the isotropic multivariate normal distribution, the maximum likelihood principle is the same as minimizing $-\log \ell(Y | \theta)$ which coincides with the mean squared error of Equation (7) (Hastie et al. 2009),

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \left\{ -\log \ell(Y|\theta) \right\} = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} \frac{1}{2\sigma_i^2} \, \| y_i - H(u(t_i;\theta)) \|_2^2. \tag{10}$$

A Bayesian formulation of equation (10) would consist in deriving a point estimate θ^*, the posterior mean of the maximum a posteriori (MAP), based on the posterior distribution for θ following Bayes theorem as $p(\theta|Y) = p(Y|\theta)\, p(\theta)/p(Y)$ where $p(\theta)$ is the prior distribution (Murphy 2022). In most realistic cases, the posterior distribution is approximated using Markov chain Monte Carlo (MCMC) sampling methods (Gelman et al. 2013). Being able to further compute gradients of the likelihood allows to design more efficient sampling methods, such as Hamiltonian Monte Carlo (Betancourt 2017).

- **A quantity of interest.** This is useful when we want to know how the solution itself changes as we move the parameter values; or more generally when it returns the value of some variable that is a function of the solution of a differential equation. The later corresponds to 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 (Giles et al. 2000a; Jameson 1988; Mohammadi et al. 2004).

In the rest of the manuscript we will use letter L to emphasize that in many cases this will be a loss function, but without loss of generality this includes the richer class of functions included in the previous examples.

3.1.3 Gradient-based optimization

In the context of optimization, the gradient of the loss allows performing gradient-based updates on the parameter θ by

$$\theta^{k+1} = \theta^k - \alpha_k \frac{dL}{d\theta^k}. \tag{11}$$

Gradient-based methods tend to outperform gradient-free optimization schemes, as they are not prone to the curse of dimensionality (Schartau et al. 2017). A direct implementation of gradient descent following Equation (11) is prone to converge to a local minimum and slows down in a neighborhood of saddle points. To address these issues, variants of this scheme employing more advanced updating strategies have been proposed (Ruder 2016). These methods include Newton-type methods (Xu et al. n.d.), quasi-Newton methods, acceleration techniques (Muehlebach et al. 2021), and natural gradient descent methods (Nurbekyan et al. 2023). For instance, ADAM is an adaptive, momentum-based algorithm that remembers the solution update at each iteration, and determines the next update as a linear combination of the gradient and the previous update (Kingma et al. 2014). ADAM been widely adopted to train highly parametrized neural networks (up to the order of 10^8 parameters (Vaswani et al. 2017)). Other widely employed algorithms are the Broyden–Fletcher–Goldfarb–Shanno (BFGS) and its limited-memory version algorithm (L-BFGS), which determine the descent direction by preconditioning the gradient with curvature information. ADAM is less prone to converge to a local minimum, while (L-)BFGS has a faster converge rate. Using ADAM for the first iterations followed by (L-)BFGS proves to be a successful strategy to minimize a loss function with best accuracy.

3.1.4 Sensitivity matrix

In general, loss functions considered are of the form $L(\theta) = L(u(\cdot, \theta), \theta)$. Using the chain rule we can derive

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial u} \frac{\partial u}{\partial \theta} + \frac{\partial L}{\partial \theta}. \tag{12}$$

Notice here the distinction between the direct derivative $\frac{d}{d\theta}$ and partial derivative $\frac{\partial}{\partial \theta}$. The two partial derivatives of the loss function on the right-hand side are usually easy to evaluate. For example, for the loss function in Equation (5) these are simply given by

$$\frac{\partial L}{\partial u} = u - u^{\text{target}}(t_1) \qquad \frac{\partial L}{\partial \theta} = 0. \tag{13}$$

In most applications, the empirical component of the loss function $L(\theta)$, that is, the part of the loss that is a function on the data, will depend on θ just through u, meaning $\frac{\partial L}{\partial \theta} = 0$. However, regularization terms added to the loss can directly depend on the parameter θ, that is $\frac{\partial L}{\partial \theta} \neq 0$. In both cases, the complicated term to compute is the matrix of derivatives $\frac{\partial u}{\partial \theta}$, usually referred to as the *sensitivity* s, and represents how much the full solution u varies as a function of the parameter θ,

$$s = \frac{\partial u}{\partial \theta} = \begin{bmatrix} \frac{\partial u_1}{\partial \theta_1} & \dots & \frac{\partial u_1}{\partial \theta_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_n}{\partial \theta_1} & \dots & \frac{\partial u_n}{\partial \theta_p} \end{bmatrix} \in \mathbb{R}^{n \times p}. \tag{14}$$

The sensitivity s defined in Equation (14) is a *Jacobian*, that is, a matrix of first derivatives of a vector-valued function. Methods involved in the calculation of s are naturally part of sensitivity methods. As mentioned earlier, most of forward sensitivity methods compute the full sensitivity matrix s, while reverse methods only deal with Jacobian-vector products (JVPs) of the form $\frac{\partial u}{\partial \theta} v$, for some vector $v \in \mathbb{R}^p$, saving unnecessary calculations at the expenses of larger memory cost. The product $\frac{\partial u}{\partial \theta} v$ is the directional derivative of the function $u(\theta)$, also known as the Gateaux derivative of $u(\theta)$ in the direction v, given by

$$\frac{\partial u}{\partial \theta} v = \lim_{h \to 0} \frac{u(\theta + hv) - u(\theta)}{h}, \tag{15}$$

representing how much the function u changes when we perturb θ in the direction of v.

3.2 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 the function $L : \mathbb{R}^p \mapsto \mathbb{R}$, we can approximate

$$\frac{dL}{d\theta_i}(\theta) = \frac{L(\theta + \varepsilon e_i) - L(\theta)}{\varepsilon} + \mathcal{O}(\varepsilon), \tag{16}$$

with e_i the i-th canonical vector and ε the stepsize. Even better, the centered difference scheme leads to

$$\frac{dL}{d\theta_i}(\theta) = \frac{L(\theta + \varepsilon e_i) - L(\theta - \varepsilon e_i)}{2\varepsilon} + \mathcal{O}(\varepsilon^2). \tag{17}$$

While Equation (16) gives the derivative to an error of magnitude $\mathcal{O}(\varepsilon)$, the centered differences schemes improves the accuracy to $\mathcal{O}(\varepsilon^2)$ (Ascher et al. 2011). Further finite difference stencils of higher order exist in the literature (Fornberg 1988).

Finite difference scheme are subject to a number of issues, related to the parameter vector dimension and rounding errors. Firstly, calculating directional derivatives requires at least one extra function evaluations per parameter dimension. For the centered differences approach in Equation (17), this requires a total of $2p$ function evaluations which demands solving the differential equation each time for a new set of parameters. Every computer ultimately stores and manipulates numbers using floating point arithmetic (Goldberg 1991). Equations (16) and (17) 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 balances these two effects is sometimes known as the *stepsize dilemma*, for which algorithms based on prior knowledge of the function to be differentiated or algorithms based on heuristic rules have been introduced (Barton 1992; Hindmarsh et al. 2005; Mathur 2012). Furthermore, numerical solutions of differential equations have errors that are typically larger than machine precision, which leads to inaccurate estimations of the gradient when ε is too small (see also 4).

Despite these caveats, finite differences can prove useful in specific contexts, such as computing Jacobian-vector products (JVPs). 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

$$Jv \approx \frac{f(u + \varepsilon v, \theta, t) - f(u, \theta, t)}{\varepsilon} \tag{18}$$

This approach is used in numerical solvers based on Krylov methods, where linear systems are solved by iteratively solving matrix-vectors products (Ipsen et al. 1998).

3.3 Automatic differentiation

Automatic differentiation (AD) is a technology that generates new code representing derivatives of a given parent code. Examples are code representing the tangent linear or adjoint operator of the parent code (Griewank et al. 2008). The names *algorithmic* and *computational* differentiation is also used in the literature, emphasizing the algorithmic rather than automatic nature of AD (Griewank et al. 2008; Margossian 2018; Naumann 2011). Any computer program implementing a given function can be reduced to a sequence of simple algebraic operations that have straightforward derivative expressions, based upon elementary rules of differentiation (Juedes 1991). The derivatives of the outputs of the computer program (dependent variables) with respect to their inputs (independent variables) are then combined using the chain rule. One advantage of AD systems is their capacity to differentiate complex programs that include control flow, such as branching, loops or recursions.

AD falls under the category of discrete methods. Depending on whether the concatenation of the elementary derivatives is done as the program is executed (from input to output) or in a later instance where we trace-back the calculation from the end (from output to input), we refer to *forward* or *reverse* mode AD, respectively. Neither forward nor reverse mode is more efficient in all cases (Griewank 1989), as we will discuss in Section 3.3.3.

3.3.1 Forward mode

Forward mode AD can be implemented in different ways depending on the data structures we use when representing a computer program. Examples of these data structures include dual numbers and computational graphs (Baydin et al. 2017). These representations are mathematically equivalent and lead to the same implementation except for details in the compiler optimizations with respect to floating point ordering.

3.3.1.1 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 a certain parameter (Clifford 1871). We define a dual number based on two variables: a *value* coordinate x_1 that carries the value of the variable and a *derivative* (also known as partial or tangent) 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

$$x_\epsilon = x_1 + \epsilon\, x_2, \tag{19}$$

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 (Karczmarczuk 1998). 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

$$x_\epsilon + y_\epsilon = (x_1 + y_1) + \epsilon\,(x_2 + y_2) \qquad x_\epsilon y_\epsilon = x_1 y_1 + \epsilon\,(x_1 y_2 + x_2 y_1). \tag{20}$$

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 that 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 of ϵ as being a differential in the Taylor series expansion, as evident in how the output of any scalar functions is extended to a dual number output:

$$\begin{aligned} f(x_1 + \epsilon x_2) &= f(x_1) + \epsilon\, x_2\, f'(x_1) + \epsilon^2 \cdot (\dots) \\ &= f(x_1) + \epsilon\, x_2\, f'(x_1). \end{aligned} \tag{21}$$

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. In Section 4 we will explore how this is implemented.

Multidimensional dual number generalize dual number to include a different dual variable ϵ_i for each variable we want to differentiate with respect to (Neuenhofen 2018; Revels et al. 2016). A multidimensional dual number is then defined as $x_\epsilon = x + \sum_{i=1}^{p} x_i \epsilon_i$, with the property that $\epsilon_i \epsilon_j = 0$ for all pairs i and j. Another extension of dual numbers that should not be confused with multidimensional dual numbers are hyper-dual numbers, which allow to compute higher-order derivatives of a function (Fike 2013).

3.3.1.2 Computational graph

A 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 as a acyclic directed graph with nodes associated to variables and edges to atomic operations (Griewank 1989; Griewank et al. 2008), known as Kantorovich graph (Kantorovich 1957) or its linearized representation via a Wengert trace/tape (Bauer 1974; Griewank et al. 2008; Wengert 1964). We can define $v_{-p+1}, v_{-p+2}, \dots, v_0 = \theta_1, \theta_2, \dots, \theta_p$ the input set of variables; $v_1, \dots, v_{m-1}$ 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 the other (a quantity known as the tangent) using the Bauer formula (Bauer 1974; Oktay et al. 2020)

$$\frac{\partial v_j}{\partial v_i} = \sum_{\substack{\text{paths } w_0 \to w_1 \to \ldots \to w_K \\ \text{with } w_0 = v_i, w_K = v_j}} \prod_{k=0}^{K-1} \frac{\partial w_{k+1}}{\partial w_k}, \tag{22}$$

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 paths, a simplification is to increasingly evaluate $j = 1, \ldots, m$ using the recursion

$$\frac{\partial v_j}{\partial v_i} = \sum_{w \text{ such that } w \to v_j} \frac{\partial v_j}{\partial w} \frac{\partial w}{\partial v_i} \tag{23}$$

Since every variable node w such that $w \to v_j$ is an edge of the computational graph has an index less than j, we can iterate this procedure as we run the computer program and solve for both the function and its derivative. 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 defined in Equation (23).

3.3.2 Reverse mode

Reverse mode AD is also known as the adjoint, or cotangent linear mode, or backpropagation in the field of machine learning. The reverse mode of automatic differentiation has been introduced in different contexts (Griewank 2012) and materializes the observation made by Phil Wolfe that if the chain rule is implemented in reverse mode, then the ratio between the computational cost of the gradient of a function and the function itself can be bounded by a constant that does not depend on the number of parameters to differentiate (Griewank 1989; Wolfe 1982), a point known as the *cheap gradient principle* (Griewank 2012). Given a directed graph of operations defined by a Wengert list, we can compute gradients of any given function in the same fashion as Equation (23) but in reverse mode as

$$\bar{v}_i = \frac{\partial \ell}{\partial v_i} = \sum_{w \text{ such that } v_i \to w} \frac{\partial w}{\partial v_i} \bar{w}. \tag{24}$$

In this context, the notation $\bar{w} = \frac{\partial L}{\partial w}$ is introduced to signify the partial derivative of the output variable, here associated to the loss function, with respect to input and intermediate variables. This derivative is often referred to as the adjoint, dual, or cotangent, and its connection with the discrete adjoint method will be made more explicitly in Section 3.9.3.

Since in reverse-mode AD the values of $\bar{w}$ are being updated in reverse order, in general we need to know the state value of all the argument variables v of w in order to evaluate the terms $\frac{\partial w}{\partial v}$. These state values (required variables) need to be either stored in memory during the evaluation of the function or recomputed on the fly in order to be able to evaluate the derivative. Checkpointing schemes exist to limit and balance the amount of storing versus recomputation (see section 4.1.2.3).

3.3.3 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 (Giering et al. 1998b). 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 is that the quantity we are interested in is described by the product of a Jacobian times a vector on the left side (VJP) or the right (JVP). Furthermore, both forward and reverse AD can be thought of as a way of computing derivatives associated with JVPs (see Equation (15)) and VJPs, respectively. In other words, given a function $g : \mathbb{R}^{d_1} \mapsto \mathbb{R}^{d_2}$ that is evaluated during the forward mode of given program, AD will carry terms of the form $Dh(x) \cdot \dot{x}$ (JVP) in forward mode and $\bar{y}^T \cdot Dh(x)$ (VJP) in reverse mode (Griewank et al. 2008).

Let us consider for example the case of a loss function $L : \mathbb{R}^p \mapsto \mathbb{R}$ taking a total of p arguments as inputs 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 $g_i : \mathbb{R}^{d_{i-1}} \mapsto \mathbb{R}^{d_i}$, where we define $d_0 = p$ for simplicity. The final gradient is computed as the chain product of vectors and Jacobians as

$$\nabla_\theta L = \nabla \ell \cdot Dg_k \cdot Dg_{k-1} \cdot \ldots \cdot Dg_2 \cdot Dg_1, \tag{25}$$

with Dg_i the Jacobian of each nested function evaluated at the intermediate values $g_{i-1} \circ g_{i-2} \circ \ldots \circ g_i(\theta)$. 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 multiplying the Jacobians forward from Dg_1 to Dg_k, or from the left side, moving backwards. The important aspect of the backwards case is that we will always be computing VJPs, since $\nabla \ell$ is a vector. Since VJPs are easier to evaluate than full Jacobians, the reverse mode will in general be faster when $1 \ll p$. This example is illustrated in 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 $\mathcal{O}(d_1 d_2 d_3)$.

It is worth noting that while more efficient methods for matrix-matrix multiplication based on Strassen's recursive algorithm and its variants exist, these are not extensively used in most scientific applications (Huang et al. 2016; Silva et al. 2018). This implies that forward AD requires a total of

$$d_2 d_1 p + d_3 d_2 p + \ldots + d_k d_{k-1} p + d_k p = \mathcal{O}(kp) \tag{26}$$

operations, while backwards mode AD requires

$$d_k d_{k-1} + d_{k-1} d_{k-2} + \ldots + d_2 d_1 + d_1 p = \mathcal{O}(k + p) \tag{27}$$

operations.

In the general case of a function $L : \mathbb{R}^p \mapsto \mathbb{R}^q$ with multiple outputs and a total of k intermediate functions, the cost of forward AD is $\mathcal{O}(pk + q)$ and the cost of reverse is $\mathcal{O}(p + kq)$. When the function to differentiate has a larger input space than output ($q \ll p$), AD in reverse mode is more efficient as it propagates the chain rule by computing VJPs. For this reason, reverse-mode AD is preferred in both modern machine learning and inverse methods. However, notice that reverse mode AD requires saving intermediate variables through the forward run in order to run backwards afterwards (Bennett 1973), leading to performance overhead that makes forward AD more efficient when $p \lesssim q$ (Baydin et al. 2017; Griewank 1989; Margossian 2018).

In practice, many AD systems are reduced to the computation of only directional derivatives (JVPs) or gradients (VJPs) (Griewank et al. 2008). Full Jacobians $J \in \mathbb{R}^{n \times p}$ (e.g., the sensitivity $s = \frac{\partial u}{\partial \theta} \in \mathbb{R}^{n \times p}$) can be fully reconstructed by the independent computation of the p columns of J via the JVPs Je_i, with $e_i \in \mathbb{R}^p$ the canonical vectors; or by the calculation of the m rows of J via the VJPs $e_j^T J$, with $e_j \in \mathbb{R}^n$. Sparse Jacobians are commonplace in large-scale nonlinear systems and discretized PDEs. When the sparsity pattern is known, they can be efficiently obtained with *colored AD* to chunk multiple JVPs or VJPs, based on the colored Jacobian (Gebremedhin et al.

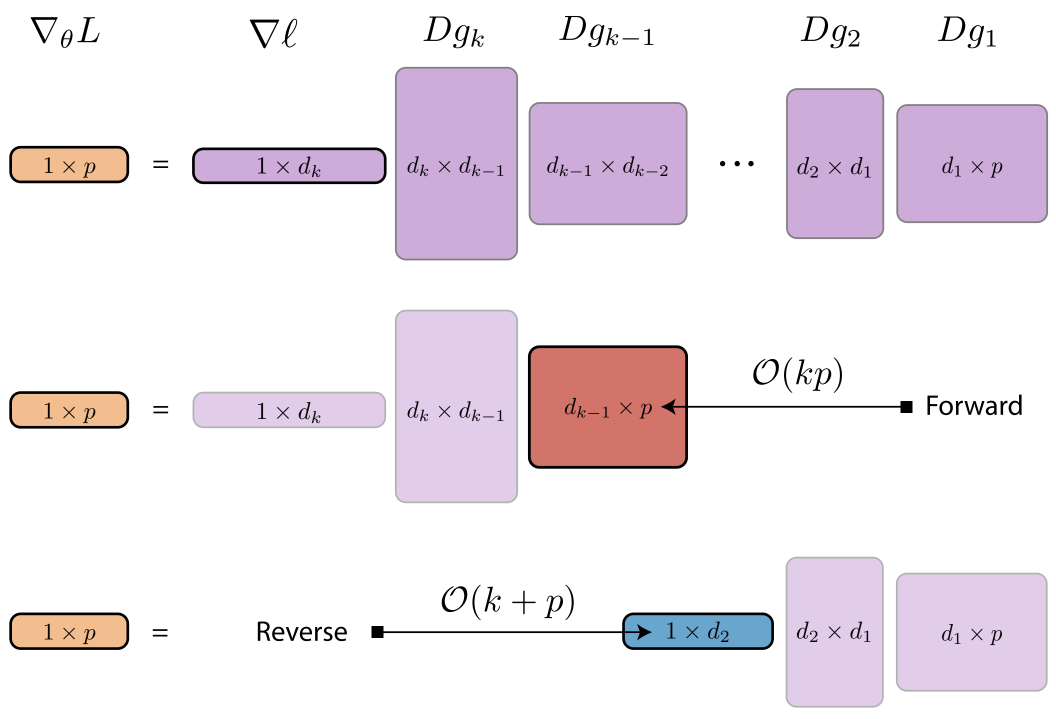

Figure 2: *Comparison between forward and reverse mode AD. Changing the order of Jacobian multiplications changes the total number of floating-point operations, which leads to different computational complexities between forward and reverse mode. When the multiplication is carried from the right side of the mathematical expression for $\nabla_\theta L$, each matrix simplification involves a matrix with size p, giving a total complexity of $\mathcal{O}(kp)$. This is the opposite of what happens when we carry the VJP from the left side of the expression, where the matrix of size $d_1 \times p$ has no effect in the intermediate calculations, making all the intermediate calculations $\mathcal{O}(1)$ with respect to p and a total complexity of $\mathcal{O}(k+p)$.*

2005). More concretely, consider the example of a Jacobian, J_{sparse}, with known sparsity pattern given by

$$J_{\text{sparse}} = \begin{bmatrix} \bullet & & & & \\ & \bullet & \bullet & & \\ & & & \bullet & \\ \bullet & \bullet & & & \bullet \\ & & & & \bullet \end{bmatrix}, \tag{28}$$

where $\bullet$ denotes the non-zero elements of the Jacobian. We can color the above matrix as follows:

$$J_{\text{sparse}}^{(\text{col})} = \begin{bmatrix} \blacktriangleright & & & & \\ & \blacksquare & \blacktriangleright & & \\ & & & \blacktriangleright & \\ \blacktriangleright & \blacksquare & & & \blacklozenge \\ & & & & \blacklozenge \end{bmatrix} \qquad J_{\text{sparse}}^{(\text{row})} = \begin{bmatrix} \blacksquare & & & & \\ & \blacksquare & \blacksquare & & \\ & & & \blacksquare & \\ \blacklozenge & \blacklozenge & & & \blacklozenge \\ & & & & \blacksquare \end{bmatrix}. \tag{29}$$

To compute $J^{(\mathrm{col})}_{\mathrm{sparse}}$, we just need to perform three JVPs,

$$J^{(\mathrm{col})}_{\mathrm{sparse}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \blacktriangleright \\ \blacktriangleright \\ \blacktriangleright \\ \blacktriangleright \\ \end{bmatrix}, \qquad J^{(\mathrm{col})}_{\mathrm{sparse}} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \\ \blacksquare \\ \\ \blacksquare \\ \end{bmatrix}, \qquad J^{(\mathrm{col})}_{\mathrm{sparse}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \\ \\ \\ \blacklozenge \\ \blacklozenge \end{bmatrix}, \tag{30}$$

compared to five JVPs for a 5×5 dense Jacobian. Similarly, since reverse mode materializes the Jacobian one row at a time, we need two VJPs (once each for ■, and ♦) compared to five VJPs for the dense counterpart.

3.4 Complex step differentiation

An alternative to finite differences that avoids subtractive cancellation 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 (Lyness 1967; Lyness et al. 1967). A newer approach recently emerged that uses the complex generalization of a real function to evaluate its derivatives (Martins et al. 2003; Squire et al. 1998). Assuming that the function $L(\theta)$ admits a holomorphic extension (that is, it can be extended to a complex-valued function that is analytical and differentiable (Stein et al. 2010)), the Cauchy-Riemann conditions can be used to evaluate the derivative with respect to one single scalar parameter $\theta \in \mathbb{R}$ as

$$\frac{dL}{d\theta} = \lim_{\varepsilon \to 0} \frac{\operatorname{Im}(L(\theta + i\varepsilon))}{\varepsilon}, \tag{31}$$

where i is the imaginary unit satisfying $i^2 = -1$. The order of this approximation can be found using the Taylor expansion of a function,

$$L(\theta + i\varepsilon) = L(\theta) + i\varepsilon \frac{dL}{d\theta} - \frac{1}{2}\varepsilon^2 \frac{d^2 L}{d\theta^2} + \mathcal{O}(\varepsilon^3). \tag{32}$$

Computing the imaginary part $\operatorname{Im}(L(\theta + i\varepsilon))$ leads to

$$\frac{dL}{d\theta} = \frac{\operatorname{Im}(L(\theta + i\varepsilon))}{\varepsilon} + \mathcal{O}(\varepsilon^2). \tag{33}$$

The method of *complex step differentiation* consists then in estimating the gradient as $\operatorname{Im}(L(\theta + i\varepsilon))/\varepsilon$ 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. However, a major limitation of this method is that it only just for locally complex analytical functions (Martins et al. 2003) and does not outperform AD. One additional limitation is that it requires the evaluation of mathematical functions with small complex values, e.g., operations such as $\sin(1 + 10^{-16} i)$, which are not necessarily always computable to high accuracy with modern math libraries. Extension to higher order derivatives can be done by introducing multicomplex variables (Lantoine et al. 2012).

3.5 Symbolic differentiation

In symbolic differentiation, functions are represented algebraically instead of algorithmically, which is why many symbolic differentiation tools are included inside computer algebra systems (CAS)

(Gowda et al. 2022). Instead of numerically evaluating the final value of a derivative, symbolic systems assign variable names, expressions, operations, and literals to *algebraic* objects. For example, the relation $y = x^2$ is interpreted as expression with two variables, x and y, and the symbolic system generates the derivative $y' = 2 \times x$ with 2 a numeric literal, $\times$ a binary operation, and x the same variable assignment as in the original expression. When the function to differentiate is large, symbolic differentiation can lead to *expression swell*, that is, exponentially large or complex symbolic expressions (Baydin et al. 2017). Simplification routines implemented in CAS may however reduce the size and complexity of algebraic expressions by finding common sub-expressions. This can make symbolic differentiation very efficient when computing derivatives multiple times and for different input values (Dürrbaum et al. 2002).

It is important to acknowledge the close relationship between AD and symbolic differentiation. There is no agreement as to whether symbolic differentiation should be classified as AD (Elliott 2018; Juedes 1991; Laue 2019) or as a different method (Baydin et al. 2017). Both are equivalent in the sense that they perform the same operations but the underlying data structure is different (Laue 2019). Here, expression swell is a consequence of the underlying representation when this does not allow for common sub-expressions. This can also be understood as if AD is symbolic differentiation performed by a compiler (Elliott 2018), meaning that different AD can be classified based on the level of integration with the underlying source language (Juedes 1991).

3.6 Sensitivity equations

An easy way to derive an expression for the sensitivity s is by deriving the sensitivity equations (Ramsay et al. 2017), a method also referred to as continuous local sensitivity analysis (CSA). If we consider the original system of ODEs given by Equation (1) and we differentiate with respect to θ, we then obtain

$$\frac{d}{d\theta}\left(\frac{du}{dt} - f(u(\theta), \theta, t)\right) = 0. \tag{34}$$

Assuming that an unique solution exists and both $\frac{\partial f}{\partial u}$ and $\frac{\partial f}{\partial \theta}$ are continuous in the neighbourhood of the solution, or under the guarantee of interchangeability of the derivatives (Gronwall 1919), for example by assuming that both $\frac{du}{dt}$ and $\frac{du}{d\theta}$ are differentiable (Palmieri et al. 2020), we can derive

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

Identifying the sensitivity matrix $s(t)$ defined in Equation (14), we obtain the *sensitivity differential equation*

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

Both the original system of n ODEs and the sensitivity equation of np ODEs are solved simultaneously, which is necessary since the sensitivity differential equation directly depends on the value of $u(t)$. This implies that as we solve the ODEs, we can ensure the same level of numerical precision for the two of them inside the numerical solver.

In contrast to the methods previously introduced, the sensitivity equations find the derivative 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. By solving the sensitivity equation and the original differential equation for $u(t)$ simultaneously, we ensure that by the end of the forward step we have calculated both $u(t)$ and $s(t)$.

3.7 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*, simultaneously with the original system of equations defined by the numerical solver. 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 (Elliott et al. 1996; Giles et al. 2000b) or in ocean state estimation (Wunsch 2008; Wunsch et al. 2007).

The idea of the adjoint method is to treat the differential equation as a constraint in an optimization problem and then differentiate an objective function subject to that constraint. Mathematically speaking, this can be treated both from a duality or Lagrangian perspective (Giles et al. 2000b). In agreement with other authors, we prefer to derive the equation using the former as it may give better insights to how the method works and it allows generalization to other user cases (Givoli 2021). The derivation of adjoint methods using the Lagrangian formulation can be found in Appendix A.

3.7.1 Adjoint state equations

The derivation of the discrete adjoint equations is carried out once the numerical scheme for solving Equation (1) has been specified. Given a discrete sequence of timesteps $t_0, t_1, \ldots, t_N$, we aim to find approximate numerical solutions $u_i \approx u(t_i; \theta)$. Any numerical solver, including the ones discussed in Section 3.1.1, can be understood as solving the (in general nonlinear) system of equations defined by $G(U;\theta) = 0$, where U is the super-vector $U = (u_1, u_2, \ldots, u_N) \in \mathbb{R}^{nN}$, and we have combined the system of equations defined by the iterative solver as $G(U;\theta) = (g_1(u_1;\theta), \ldots, g_N(u_N;\theta)) = 0$ (see Equation (4)).

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

$$L(U,\theta) = \frac{1}{2} \sum_{i=1}^{N} \| u_i - u_i^{\text{obs}} \|^2, \tag{37}$$

with u_i^{obs} the observed time-series. Similarly to Equation (12) we have

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial U} \frac{\partial U}{\partial \theta}. \tag{38}$$

By differentiating the constraint $G(U;\theta) = 0$, we obtain

$$\frac{dG}{d\theta} = \frac{\partial G}{\partial \theta} + \frac{\partial G}{\partial U} \frac{\partial U}{\partial \theta} = 0, \tag{39}$$

which is equivalent to

$$\frac{\partial U}{\partial \theta} = - \left(\frac{\partial G}{\partial U} \right)^{-1} \frac{\partial G}{\partial \theta}. \tag{40}$$

Replacing this last expression into Equation (38), we obtain

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \frac{\partial L}{\partial U} \left(\frac{\partial G}{\partial U} \right)^{-1} \frac{\partial G}{\partial \theta}. \tag{41}$$

Importantly, the right-hand side of Equation (41) can be resolve as a vector-Jacobian product (VJP), with $\frac{\partial L}{\partial U}$ being the vector. Instead of computing the product of the matrices $\left(\frac{\partial G}{\partial U}\right)^{-1}$ and

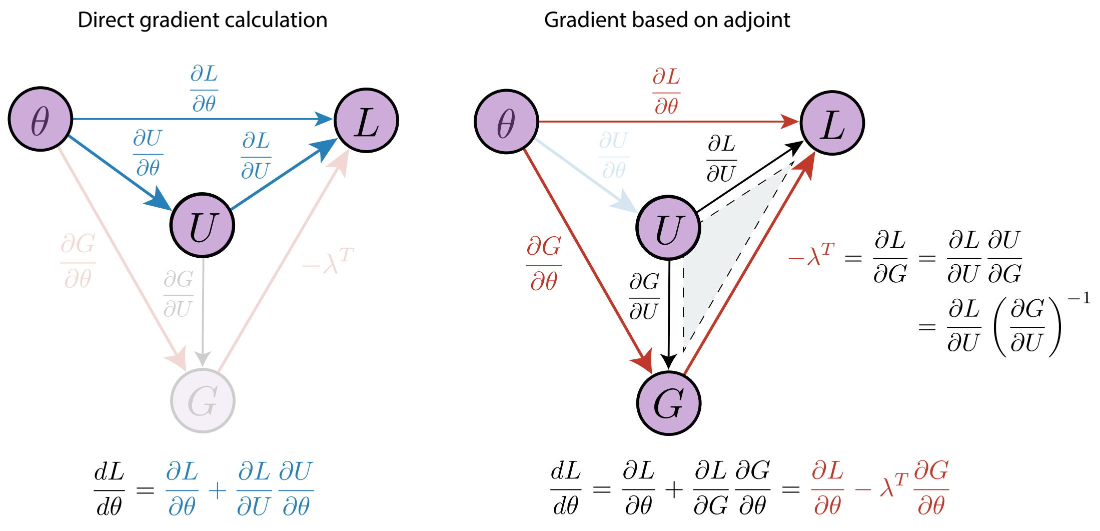
Direct gradient calculation
Gradient based on adjoint
θ
L
U
G
$\frac{\partial L}{\partial \theta}$
$\frac{\partial U}{\partial \theta}$
$\frac{\partial L}{\partial U}$
$\frac{\partial G}{\partial \theta}$
$\frac{\partial G}{\partial U}$
$-\lambda^T$
$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial \theta}$
$-\lambda^T = \frac{\partial L}{\partial G} = \frac{\partial L}{\partial U}\frac{\partial U}{\partial G} = \frac{\partial L}{\partial U}\left(\frac{\partial G}{\partial U}\right)^{-1}$
$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial G}\frac{\partial G}{\partial \theta} = \frac{\partial L}{\partial \theta} - \lambda^T \frac{\partial G}{\partial \theta}$

Figure 3: *Diagram showing how gradients are computed using discrete adjoints. On the left, we see how gradients will be computed if we use the chain rule applied to the directed triangle defined by the variables θ, U, and L (blue arrows). However, we can define the intermediate vector variable $G = G(U;\theta)$, which satisfies $G = 0$ as long as the discrete system of differential equations are satisfied, and apply the chain rule instead to the triangle defined by θ, G, and L (red arrows). In the red diagram, the calculation of $\frac{\partial L}{\partial G}$ is done by pivoting in U as shown in the right diagram (shaded area). Notice that the use of adjoints avoids the calculation of the sensitivity $\frac{\partial U}{\partial \theta}$. The adjoint is defined as the partial derivative $\lambda^T = -\frac{\partial L}{\partial G}$ representing changes in the loss function due to variations in the discrete equation $G(U;\theta) = 0$.*

$\frac{\partial G}{\partial \theta}$, it is computationally more efficient first to compute the resulting vector from the VJP 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 leads to the definition of the adjoint $\lambda \in \mathbb{R}^{nN}$ as the solution of the linear system of equations

$$\left(\frac{\partial G}{\partial U}\right)^T \lambda = \left(\frac{\partial L}{\partial U}\right)^T, \tag{42}$$

or equivalently,

$$\lambda^T = \frac{\partial L}{\partial U}\left(\frac{\partial G}{\partial U}\right)^{-1}. \tag{43}$$

Replacing Equation (43) into (41) yields

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \lambda^T \frac{\partial G}{\partial \theta}. \tag{44}$$

The important trick used in the adjoint method is the rearrangement of the multiplicative terms involved in equation (41). 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 and relying on the intermediate variable $G(U;\theta)$, we can make the same computation more efficient. These ideas are summarized in the diagram in Figure 3, where we can also see an interesting interpretation of the adjoint as being equivalent to $\lambda^T = -\frac{\partial L}{\partial G}$.

Notice that the algebraic equation of the adjoint λ in Equation (42) is a linear system of equations, even when the original system $G(U;\theta) = 0$ is not necessarily linear in U. This means that while solving the original system of ODEs may require multiple iterations in order to solve the non-linear system $G(U) = 0$ (e.g., by using Krylov methods), the backwards step to compute the adjoint is one single linear system of equations.

3.7.2 Simple linear system

To gain further intuition about the discrete adjoint method, let us consider the simple case of the explicit linear one-step methods, where at every step we need to solve the equation $u_{i+1} = g_i(u_i;\theta) = A_i(\theta)\, u_i + b_i(\theta)$, where $A_i(\theta) \in \mathbb{R}^{n\times n}$ and $b_i(\theta) \in \mathbb{R}^n$ are defined by the numerical solver (Johnson 2012). This condition can be written in a more compact manner as $G(U) = A(\theta)U - b(\theta) = 0$, that is

$$A(\theta)U = \begin{bmatrix} \mathbb{I}_n & 0 & & & \\ -A_1 & \mathbb{I}_n & 0 & & \\ & -A_2 & \mathbb{I}_n & 0 & \\ & & & \ddots & \\ & & & -A_{N-1} & \mathbb{I}_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} A_0 u_0 + b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-1} \end{bmatrix} = b(\theta), \tag{45}$$

with $\mathbb{I}_n$ the identity matrix of size $n \times n$. Notice that in most cases, the matrix $A(\theta)$ is quite large and mostly sparse. While this representation of the discrete differential equation is convenient for mathematical manipulations, when solving the system we rely on iterative solvers that save memory and computation.

For the linear system of discrete equations $G(U;\theta) = A(\theta)U - b(\theta) = 0$, we have

$$\frac{\partial G}{\partial \theta} = \frac{\partial A}{\partial \theta} U - \frac{\partial b}{\partial \theta}, \tag{46}$$

so the desired gradient in Equation (44) can be computed as

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \lambda^T \left(\frac{\partial A}{\partial \theta} U - \frac{\partial b}{\partial \theta} \right) \tag{47}$$

with λ the discrete adjoint obtained by solving the linear system in Equation (42),

$$A(\theta)^T \lambda = \begin{bmatrix} \mathbb{I}_n & -A_1^T & & & \\ 0 & \mathbb{I}_n & -A_2^T & & \\ & 0 & \mathbb{I}_n & -A_3^T & \\ & & & \ddots & -A_{N-1}^T \\ & & & 0 & \mathbb{I}_n \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \vdots \\ \lambda_N \end{bmatrix} = \begin{bmatrix} u_1 - u_1^{\text{obs}} \\ u_2 - u_2^{\text{obs}} \\ u_3 - u_3^{\text{obs}} \\ \vdots \\ u_N - u_N^{\text{obs}} \end{bmatrix} = \frac{\partial L}{\partial U}^T. \tag{48}$$

This is a linear system of equations with the same size as 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 at the same computational cost (e.g., by using the LU factorization of $A(\theta)$). Another more natural way of finding the adjoints λ is by noticing that the system of equations (48) is equivalent to the final value problem

$$\lambda_i = A_i^T \lambda_{i+1} + (u_i - u_i^{\text{obs}}) \tag{49}$$

with final condition λ_N. This means that we can solve the adjoint equation backwards, i.e., in reverse mode, starting from the final state λ_N and computing the values of λ_i in decreasing index order. Unless the loss function L is linear in U, this procedure requires knowledge of the value of u_i (or some equivalent form of it) at any given timestep.

3.8 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. We encourage the interested reader to make the effort of following how the continuous adjoint method follows the same logic as the discrete adjoint method, 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. The Lagrangian derivation of the continuous adjoint method can also be found in Appendix A.

Consider an integrated loss function defined in Equation (8) of the form

$$L(u;\theta) = \int_{\mathrm{t}_0}^{\mathrm{t}_1} h(u(t;\theta),\theta)dt \tag{50}$$

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

$$\frac{dL}{d\theta} = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(\frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u} s(t) \right) dt. \tag{51}$$

Just as in the case of the discrete adjoint method, the complicated term to evaluate in the last expression is the sensitivity (Equation (14)). Again, the trick is to evaluate the VJP $\frac{\partial h}{\partial u}\frac{\partial u}{\partial \theta}$ by first 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

$$\int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \left(\frac{ds}{dt} - \frac{\partial f}{\partial u}\, s - \frac{\partial f}{\partial \theta} \right) dt = 0, \tag{52}$$

where this equation must be satisfied for every function $\lambda(t)$ in order for Equation (36) to be true. The next step is to get rid of the time derivative applied to the sensitivity $s(t)$ using integration by parts:

$$\int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \frac{ds}{dt} dt = \lambda(t_1)^T s(t_1) - \lambda(t_0)^T s(t_0) - \int_{\mathrm{t}_0}^{\mathrm{t}_1} \frac{d\lambda^T}{dt} s(t)\, dt. \tag{53}$$

Replacing this last expression into Equation (52) we obtain

$$\int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(- \frac{d\lambda^T}{dt} - \lambda(t)^T \frac{\partial f}{\partial u} \right) s(t) dt = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt - \lambda(t_1)^T s(t_1) + \lambda(t_0)^T s(t_0). \tag{54}$$

At first glance, there is nothing particularly interesting about this last equation. However, both Equations (51) and (54) involve a VJP with $s(t)$. Since Equation (54) must hold for every function $\lambda(t)$, we can pick $\lambda(t)$ to make the terms involving $s(t)$ in Equations (51) and (54) to perfectly coincide. This is done by defining the adjoint $\lambda(t)$ to be the solution of the new system of differential equations

$$\frac{d\lambda}{dt} = - \frac{\partial f}{\partial u}^T \lambda - \frac{\partial h^T}{\partial u} \qquad \lambda(t_1) = 0. \tag{55}$$

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 (54) to

$$\int_{\mathrm{t}_0}^{\mathrm{t}_1} \frac{\partial h}{\partial u} s(t) dt = \lambda(t_0)^T s(t_0) + \int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt. \tag{56}$$

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

$$\frac{dL}{d\theta} = \lambda(t_0)^T s(t_0) + \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta} \right) dt \tag{57}$$

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

(i) Solve the original differential equation $\frac{du}{dt} = f(u, t, \theta)$;

(ii) Solve the backwards adjoint differential equation given by Equation (55);

(iii) Compute the gradient using Equation (57).

The same recipe used here to derived the continuous adjoint method for a system of ODEs can be employed to derive adjoint methods for PDEs (Giles et al. 2000b) and DAE (Cao et al. 2002).

3.9 Mathematical comparison of the methods

In Sections 3.2 – 3.8 we focused on merely introducing each one of the sensitivity methods classified in Figure 1 as separate methods, postponing the discussion about their points in common. In this section, we compare one-to-one these methods and highlight differences and parallels between them. We hope this enlightens the discussion on sensitivity methods and helps to demystify misconceptions around the sometimes apparent differences between methods. This comparison will be strengthened again later in Section 4, where we will see how even small differences between methods can be translated to different software implementations with different advantages.

3.9.1 Forward AD and complex step differentiation

Both AD based on dual numbers and complex-step differentiation introduce an abstract unit (ϵ and i, respectively) associated with the imaginary part of the dual variable that carries forward the numerical value of the gradient. Although these methods seem similar, we emphasize that AD gives the exact gradient value, whereas complex step differentiation relies on numerical approximations that are valid only when the stepsize ε is small. In Figure 4 we show how the calculation of the gradient of the function $\sin(x^2)$ is performed by these two methods. Whereas the second component of the dual number has the exact derivative of the function $\sin(x^2)$ with respect to x, it is not until we take $\varepsilon \to 0$ that we obtain the derivative in the imaginary component for the complex step method. The dependence of the complex step differentiation method on the step size gives it a closer resemblance to finite difference methods than to AD using dual numbers. Further notice the complex step method involves more terms in the calculation, a consequence of the fact that second order terms of the form $i^2 = -1$ are transferred to the real part of the complex number, while for dual numbers the terms associated to $\epsilon^2 = 0$ vanish (Martins et al. 2001).

3.9.2 AD, finite differences and symbolic differentiation with sparsity

When sparsity patterns of the gradient/Jacobian to be computed are known, symbolic differentiation can be more efficient than AD and finite differences (Lantoine et al. 2012). In Section 3.3.3 we discussed how colored AD can be used to efficiently compute a sparse Jacobian. However, colored AD has the limitation that an extremely sparse matrix can have no rows or columns independent

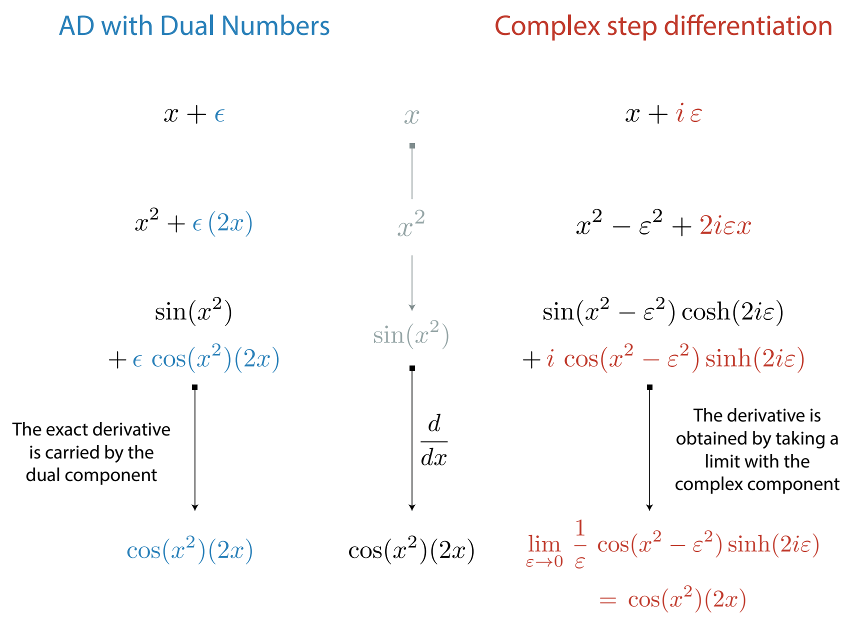
AD with Dual Numbers
Complex step differentiation
$x+\epsilon$
x
$x+i\varepsilon$
$x^2+\epsilon(2x)$
x^2
$x^2-\varepsilon^2+2i\varepsilon x$
$\sin(x^2)$
$+\epsilon\cos(x^2)(2x)$
$\sin(x^2)$
$\sin(x^2-\varepsilon^2)\cosh(2i\varepsilon)$
$+i\cos(x^2-\varepsilon^2)\sinh(2i\varepsilon)$
The exact derivative is carried by the dual component
$\frac{d}{dx}$
The derivative is obtained by taking a limit with the complex component
$\cos(x^2)(2x)$
$\cos(x^2)(2x)$
$\lim_{\varepsilon\to 0}\frac{1}{\varepsilon}\cos(x^2-\varepsilon^2)\sinh(2i\varepsilon)$
$=\cos(x^2)(2x)$

Figure 4: *Comparison between AD implemented with dual numbers and complex step differentiation. For the simple case of the function $f(x)=\sin(x^2)$, we can see how each operation is carried in the forward step by the dual component (blue) and the complex component (red). Whereas AD gives the exact gradient at the end of the forward run, in the case of the complex step method we need to take the limit in the imaginary part.*

of each other. Consider the arrowhead matrix given by

$$J_{\text{arrowhead}} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & & & \\ \bullet & & \bullet & & \\ \bullet & & & \bullet & \\ \bullet & & & & \bullet \end{bmatrix}. \tag{58}$$

In this case, both reverse mode AD and forward mode AD will have to perform n VJPs and JVPs, respectively (for $J_{\text{arrowhead}} \in \mathbb{R}^{n\times n}$ arrowhead matrix), and there is no computational benefit of coloring the matrix here. Instead, symbolic differentiation constructs a symbolic representation of the sparse Jacobian and can fill the Jacobian with $n + 2\cdot(n-1)$ computations, where each computation is significantly cheaper than each VJP or JVP.

3.9.3 Discrete adjoints and reverse AD

Both discrete adjoint methods and reverse AD are classified as discrete and reverse methods (see Figure 1). Furthermore, both methods introduce an intermediate adjoint associated with the partial derivative of the loss function (output variable) with respect to intermediate variables of the forward computation. In the case of reverse AD, this adjoint is defined with the notation $\bar{w}$ (Equation (24)), while in the discrete adjoint method this correspond to each one of the variables $\lambda_1, \lambda_2, \ldots, \lambda_N$ (Equation (48)). In this section we will show that both methods are mathematically equivalent (Li et al. 2020a; Zhu et al. 2021), but naive implementations using reverse AD can result in sub-optimal

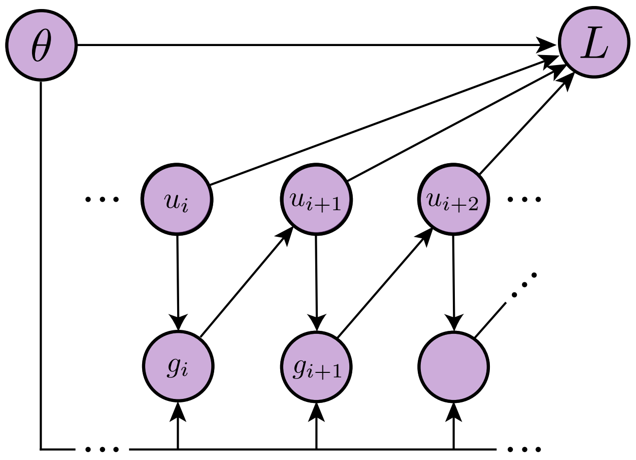

Figure 5: *Computational graph associated to the discrete adjoint method. Reverse AD applied on top of the computational graph leads to the update rules for the discrete adjoint. The adjoint variable λ_i in the discrete adjoint method coincides with the adjoint variable $\bar{g}_i$ defined in the backpropagation step.*

performances compared to the one obtained by directly employing the discrete adjoint method (Alexe et al. 2009).

In order to have a better idea of how this works in the case of a numerical solver, let us consider again the case of a one-step explicit method, not necessarily linear, where the updates u_i satisfy the equation $u_{i+1} = g_i(u_i, \theta)$. Following the same schematics as in Figure 3, we represent the computational graph of the numerical method using the intermediate variables $g_1, g_2, \ldots, g_{N-1}$. The dual/adjoint variables defined in reverse AD in this computational graph are given by

$$\bar{g}_i = \frac{\partial u_{i+1}}{\partial g_i} \bar{u}_{i+1} = \frac{\partial L}{\partial u_{i+1}} + \frac{\partial g_{i+1}}{\partial u_i} \bar{g}_{i+1}. \tag{59}$$

The updates of $\bar{g}_i$ then mathematically coincide with the updates in reverse mode of the adjoint variable λ_i (see Equation (49)).

Modern numerical solvers use functions g_i that correspond to nested functions, meaning $g_i = g_i^{(k_i)} \circ g_i^{(k_i-1)} \circ \ldots \circ g_i^{(1)}$. This is certainly the case for implicit methods when u_{i+1} is the solution of an iterative Newton method (Hindmarsh et al. 2005); or in cases where the numerical solver includes internal iterative sub-routines (Alexe et al. 2009). If the number of intermediate function is large, reverse AD will result in a large computational graph, potentially leading to excessive memory usage and slow computation (Alexe et al. 2009; Margossian 2018). A solution to this problem is to introduce a customized *super node* that directly encapsulates the contribution to the full adjoint in g_i without computing the adjoint for each intermediate function $g_i^{(j)}$. Provided with the value of the Jacobian matrices $\frac{\partial g_i}{\partial u_i}$ and $\frac{\partial g_i}{\partial \theta}$, we can use the implicit function theorem to find the $\frac{\partial u_i}{\partial \theta}$ as the solution of the linear system of equations

$$\frac{\partial g_i}{\partial u_i} \frac{\partial u_i}{\partial \theta} = -\frac{\partial g_i}{\partial \theta} \tag{60}$$

and implement AD by directly solving this new system of equations (Bell et al. 2008; Christianson 1994, 1998). In both cases, the discrete adjoint method can be implemented directly on top of a reverse AD tool that allows customized adjoint calculation (Rackauckas et al. 2021).

3.9.4 Consistency: forward AD and sensitivity equations

The sensitivity equations can also be solved in discrete forward mode by numerically discretizing the original ODE and later deriving the discrete sensitivity equations (Ma et al. 2021). For most cases, this leads to the same result as in the continuous case (Zhang et al. 2014). We can numerically solve for the sensitivity s by extending the parameter θ to a multidimensional dual number

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_p \end{bmatrix} \longrightarrow \begin{bmatrix} \theta_1 + \epsilon_1 \\ \theta_2 + \epsilon_2 \\ \vdots \\ \theta_p + \epsilon_p \end{bmatrix} \tag{61}$$

where $\epsilon_i \epsilon_j = 0$ for all pairs of i and j (see Section 3.3.1.1). The dependency of the solution u of the ODE on the parameter θ is now expanded following Equation (21) as

$$u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \longrightarrow \begin{bmatrix} u_1 + \sum_{j=1}^p \frac{\partial u_1}{\partial \theta_j} \epsilon_j \\ u_2 + \sum_{j=1}^p \frac{\partial u_2}{\partial \theta_j} \epsilon_j \\ \vdots \\ u_p + \sum_{j=1}^p \frac{\partial u_n}{\partial \theta_j} \epsilon_j \end{bmatrix} = u \, + \, s \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_p \end{bmatrix}, \tag{62}$$

that is, the dual component of the vector u corresponds exactly to the sensitivity matrix s. This implies forward AD applied to any multistep linear solver will result in the application of the same solver to the sensitivity equation (Equation (36)). For example, for the forward Euler method this gives

$$\begin{aligned} u^{t+1} + s^{t+1}\,\epsilon &= u^t + s^t\,\epsilon + \Delta t\, f(u^t + s^t\,\epsilon, \theta + \epsilon, t) \\ &= u^t + f(u^t, \theta, t) + \Delta t \left(\frac{\partial f}{\partial u} s^t + \frac{\partial f}{\partial \theta} \right) \epsilon. \end{aligned} \tag{63}$$

The dual component corresponds to the forward Euler discretization of the sensitivity equation, with s^t the temporal discretization of the sensitivity $s(t)$.

The consistency result for discrete and continuous methods also holds for Runge-Kutta methods (Walther 2007). When introducing dual numbers, the Runge-Kutta scheme in Equation (2) gives the following identities

$$u^{n+1} + s^{n+1}\epsilon = s_n + \Delta t_n \sum_{i=1}^{s} b_i \dot{k}_i \tag{64}$$

$$k_i + \dot{k}_i \epsilon = f\left(u^n + \sum_{j=1}^{s} a_{ij} k_j + \left(s^n + \sum_{j=1}^{s} a_{ij} \dot{k}_j \right) \epsilon, \theta + \epsilon, t_n + c_i \Delta t_n \right) \tag{65}$$

with $\dot{k}_i$ the dual variable associated to k_i. The partial component in Equation (65) carrying the coefficient ϵ gives

$$\begin{aligned} \dot{k}_i = &\frac{\partial f}{\partial u} \left(u^n + \sum_{j=1}^{s} a_{ij} k_j, \theta, t_n + c_i \Delta t_n \right) \left(s^n + \sum_{j=1}^{s} a_{ij} \dot{k}_j \right) \\ &+ \frac{\partial f}{\partial \theta} \left(u^n + \sum_{j=1}^{s} a_{ij} k_j, \theta, t_n + c_i \Delta t_n \right), \end{aligned} \tag{66}$$

which coincides with the Runge-Kutta scheme we would obtain for the original sensitivity equation. This means that forward AD on Runge-Kutta methods leads to solutions for the sensitivity that have the same convergence properties of the forward solver.

Note that consistency does not imply that an ODE solver is necessarily correct or stable under such a transformation. Such properties involve other aspects of the solver, such as adaptivity and error control, take into account the properties of the derivative. In Appendix B, we demonstrate that common implementations of adaptive ODE solvers may not be convergent when forward AD is applied to solver even though the process is mathematically consistent. This highlights that additional factors beyond consistency must be considered when investigating whether an implementation is convergent.

3.9.5 Consistency: discrete and continuous adjoints

As previously mentioned, the difference between the discrete and continuous adjoint methods is that the former follows the discretize-then-differentiate approach (also known as finite difference of adjoints (Sirkes et al. 1997)). In contrast, continuous adjoint equations are derived by directly operating on the differential equation, without a priori consideration of the numerical scheme used to solve it. In some sense, we can think of the discrete adjoint $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_N)$ in Equation (48) as the discretization of the continuous adjoint $\lambda(t)$.

A natural question then is if these two methods effectively compute the same gradient, i.e., if the discrete adjoint consistently approximate its continuous counterpart. Furthermore, since the continuous adjoint method requires to numerically solve the adjoint, we are interested in the relative accuracy of the forward and reverse step. It has been shown that for both explicit and implicit Runge-Kutta methods, as long as the coefficients in the numerical scheme given in Equation (2) satisfy the condition $b_i \neq 0$ for all $i = 1, 2, \ldots, s$, then the discrete adjoint is a consistent estimate of the continuous adjoint with same level of convergence as for the forward numerical solver (Hager 2000; Sandu 2006, 2011; Walther 2007). To guarantee the same order of convergence, it is important that both the forward and backward solver use the same Runge-Kutta coefficients (Alexe et al. 2009). Importantly, even when consistent, the code generated using the discrete adjoint using AD tools (see Section 3.9.3) can be sub-optimal and manual modification of the differentiation code are required to guarantee correctness (Alexe et al. 2007; Eberhard et al. 1996).

4 Implementation: A computer science perspective

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 between the methods introduced in Section 3, i.e., between those that apply direct differentiation at the algorithmic level and those that are based on numerical solvers. The former require a much different implementation since they are agnostic with respect to the mathematical and numerical properties of the system of ODEs. The latter family of methods that are based on numerical solvers include the sensitivity equations and the adjoint methods, both discrete and continuous. This section is then divided in two parts:

- **Direct methods.** (Section 4.1) Their implementation occurs at a higher hierarchy than the numerical solver software. They include finite differences, AD, complex step differentiation.

- **Solver-based methods.** Their implementation occurs at the same level of the numerical solver. They include

– Sensitivity equations (Section 4.2.1)
– Adjoint methods, both discrete and continuous (Section 4.2.2)

While these methods can be 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 (Bezanson et al. 2017, 2012), which a strong emphasis in differential equation solvers (Rackauckas et al. 2016) and sensitivity analysis (Rackauckas et al. 2020). Nevertheless, in reviewing existing work, we will also point to applications developed in other programming languages.

The GitHub repository `DiffEqSensitivity-Review` contains both text and code used to generate this manuscript. See Appendix C for a complete description of the scripts provided. The symbol ♣ will be use to reference code generated figures.

4.1 Direct methods

Direct methods are implemented independent of the structure of the differential equation and the numerical solver used to solve it. These include finite differences, complex step differentiation, and both forward and reverse mode AD.

4.1.1 Finite differences

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`, which already include subroutines to determine step-sizes. However, finite differences are less accurate and as costly as forward AD (Griewank 1989) and complex-step differentiation. Figure 6 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. Here we consider the solution of the differential equation $u'' + \omega^2 u = 0$ with initial condition $u(0) = 0$ and $u'(0) = 1$, which has analytical solution $u_{\text{true}}(t) = \sin(\omega t)/\omega$. The numerical solution $u_{\text{num}}(t)$ can be founded by solving the system of ODEs

$$\begin{cases} \frac{du_1}{dt} = u_2 & u_1(0) = 0 \\ \frac{du_2}{dt} = -\omega^2 u_1 & u_2(0) = 1. \end{cases} \tag{67}$$

The loss function used to differentiate is given by $L(\theta) = u(10)$. We see that finite differences are inaccurate for computing the derivative of u_{true} with respect to ω (that is, $u'_{\text{true}} = \cos(\omega t) - \sin(\omega t)/\omega^2$) when the stepsize ε is both too small and too large (red dashed line). When the derivative is instead computed using the numerical solution $u_{\text{num}}(t)$ (red circles), the accuracy of the derivative further deteriorates due to approximation errors in the solver. This effect is dependent on the numerical solver tolerance. Notice that in general we do not know the true solution of the differential equation, so the derivative of u_{true} obtained using finite differences just serves as a lower bound of the error we expect to see when performing sensitivity analysis on top of the numerical solver.

4.1.2 Automatic differentiation

The AD algorithms described in Section 3.3 implement algorithmic differentiation using different methods, namely *operator overloading* for AD based on dual numbers, and *source code transformation* for both forward and reverse AD based on the computational graph (Martins et al. 2001). In

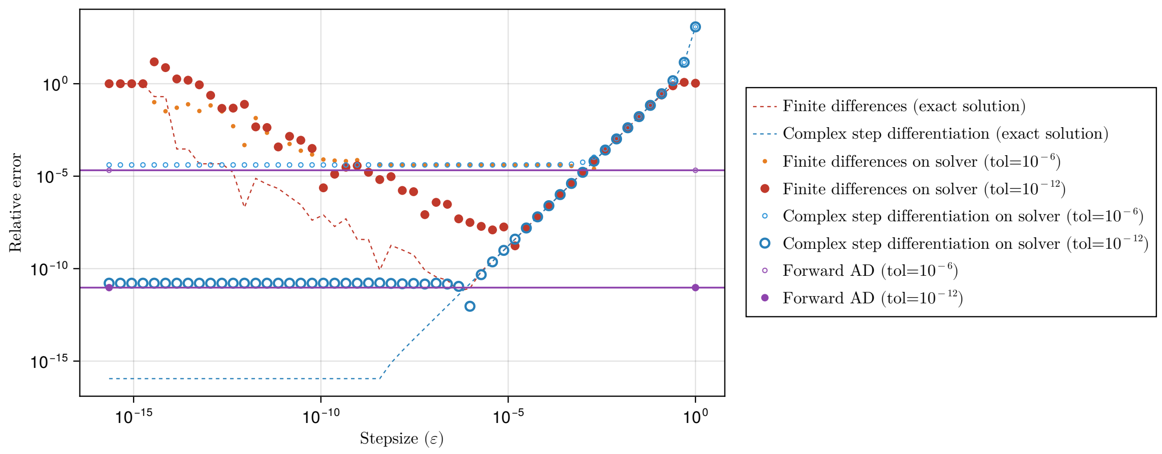

Figure 6: *Absolute relative error when computing the gradient of the function $u(t) = \sin(\omega t)/\omega$ with respect to ω at $t = 10.0$ as a function of the stepsize ε. Here, $u(t)$ corresponds to the solution of the differential equation $u'' + \omega^2 u = 0$ with initial condition $u(0) = 0$ 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 (Tsitouras 2011) 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.* $\clubsuit_1$

this section we cover how forward AD is implemented using dual numbers, postponing the implementation of AD based on computational graphs for reverse AD (Section 4.1.2.2).

4.1.2.1 Forward AD based on dual numbers

Implementing forward AD using dual numbers is usually carried out using *operator overloading* (Neuenhofen 2018). This means expanding the object associated with a numerical value to include the tangent and extending the definition of atomic algebraic functions. In Julia, this can be done by relying on multiple dispatch (Bezanson et al. 2017). The following example illustrates how to define a dual number and its associated binary addition and multiplication extensions $\clubsuit_2$.

```julia
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 (21):

```
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 (Revels et al. 2016) and the sensitivity method `ForwardDiffSensitivity` implements forward differentiation inside the numerical solver using dual numbers. Figure 6 shows the result of performing forward AD inside the numerical solver. We can see that for this simple example forward AD performs as good as the best output of finite differences and complex step differentiation (see Section 4.1.3) when optimizing by the stepsize ε. Further notice that AD is not subject to same level of numerical errors due to floating point arithmetic (Griewank et al. 2008).

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 derivative calculations (Griewank 1989). This consideration also applies to the other forward methods, including finite differences and complex-step differentiation.

Although AD is always algorithmically correct, when combined with a numerical solver *AD can be numerically incorrect* and result in wrong gradient calculations (Eberhard et al. 1996). To illustrate this point, consider the following example of a simple system of ODEs

$$\begin{cases} \frac{du_1}{dt} = au_1 - u_1 u_2 & \qquad u_1(0) = 1 \\ \frac{du_2}{dt} = -au_2 + u_1 u_2 & \qquad u_2(0) = 1 \end{cases} \tag{68}$$

with a the parameter with respect to which we want to differentiate. In the simple case of $a = 1$, the solutions of the ODE are constant functions $u_1(t) \equiv u_2(t) \equiv 1$. As explained in Appendix B, traditional adaptive stepsize solvers used for just solving ODEs are designed to control for numerical errors in the ODE solution but not in its sensitivities when coupled with an internal AD method. This can lead to wrong gradient calculations that propagate through the numerical solver without further warning $\clubsuit_3$.

4.1.2.2 Reverse AD based on computational graph

In contrast to finite differences, forward AD, and complex-step differentiation, reverse AD is the only of this family of methods that propagates the gradient in reverse mode by relying on analytical derivatives of primitive functions, which in Julia are available via `ChainRules.jl` Since this requires the evaluation intermediate variables, reverse AD requires a more delicate protocol of how to store intermediate variables in memory and make them accessible during the backwards pass.

Reverse AD can be implemented via *pullback* functions (Innes 2018), a method also known as *continuation-passing style* (Wang et al. 2019). In the backward step, it 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 (defined as $\bar{v}_i$) by applying the chain rule. Consider the example of the multiplicative operation $w = v_1 \times v_2$. Then

$$\bar{v}_1, \, \bar{v}_2 \;=\; v_2 \times \bar{w}, \, v_1 \times \bar{w} \;=\; \mathcal{B}_w(\bar{w}), \tag{69}$$

which is equivalent to using the chain rule as

$$\frac{\partial \ell}{\partial v_1} = \frac{\partial}{\partial v_1}(v_1 \times v_2)\frac{\partial \ell}{\partial w}. \tag{70}$$

A crucial distinction between AD implementations based on computational graphs is between *static* and *dynamic* methods (Baydin et al. 2017). In the case of a static implementations, the computational graph is constructed before any code is executed, which are encoded and optimized for performance within the graph language. For static structures such as neural networks, this is ideal (Abadi et al. 2016). However, two major drawbacks of static methods are composability with existing code, including support of custom types, and adaptive control flow, which is a common feature of numerical solvers. These issues are addressed in reverse AD implementations using *tracing*, where the program structure is transformed into a list of pullback functions that build the graph dynamically at runtime. Popular libraries in this category are `Tracker.jl` and `ReverseDiff.jl`. A major drawback of tracing systems is that the pullbacks are constructed with respect to the control flow of the input value and thus do not necessarily generalize to other inputs. This means that the pullback must be reconstructed for each now forward pass, limiting the reuse of computational optimizations or inducing higher overhead. Source-to-source AD systems can achieve higher performance by giving a static derivative representation to arbitrary control flow structure, thus allowing for the construction and optimization of pullbacks independent of the input value. These include `Zygote.jl` (Innes et al. 2019), `Enzyme.jl` (Moses et al. 2020; Moses et al. 2021), and `Diffractor.jl`. The existence of multiple AD packages lead to the development of `AbstractDifferentiation.jl` which allows to combine different methods (Schäfer et al. 2021b).

It is important to mention that incorrect implementations of both forward and reverse AD can lead to *perturbation confusion*, an existing problem in some AD software where either repeated applications of AD or differentiation with respect to different dual variables result indistinguishable (Manzyuk et al. 2019; Siskind et al. 2005).

4.1.2.3 Discrete checkpointing

In contrast to forward methods, all reverse methods, including backpropagation and adjoint methods, require to access the value of intermediate variables during the propagation of the gradient. For a numerical solver, the number of memory required to accomplish this can be very large, involving a total of at least $\mathcal{O}(nk)$ terms, with k the number of steps of the numerical solver. Checkpointing is a technique that can be used for all reverse methods. It avoids storing all the intermediate states by balancing storing and recomputation to recover the required state exactly (Griewank et al. 2008). This is achieved by saving intermediate states of the solution in the forward pass and recalculating the solution between intermediate states in the backwards mode (Griewank et al. 2008). Different checkpointing algorithms have been proposed, ranging from static, multi-level (e.g., (Giering et al. 1998a; Heimbach et al. 2005) to optimized, binomial checkpointing algorithms (e.g., (**Walther:2004**; Schanen et al. 2023)).

4.1.3 Complex step differentiation

Modern software already have support for complex number arithmetic, making complex step differentiation very easy to implement. In Julia, complex analysis arithmetic can be easily carried inside the numerical solver. The following example shows how to extend the numerical solver used to solve the ODE in Equation (67) to support complex numbers $\clubsuit_4$.

```
function dyn!(du::Array{Complex{Float64}}, u::Array{Complex{Float64}}, p, t)
    ω = p[1]
    du[1] = u[2]
    du[2] = - ω^2 * u[1]
end

tspan = [0.0, 10.0]
du = Array{Complex{Float64}}([0.0])
u0 = Array{Complex{Float64}}([0.0, 1.0])

function complexstep_differentiation(f::Function, p::Float64, ε::Float64)
    p_complex = p + ε * im
    return imag(f(p_complex)) / ε
end

complexstep_differentiation(x -> solve(ODEProblem(dyn!, u0, tspan, [x]), Tsit5()
    ).u[end][1], 20., 1e-3)
```

Figure 6 further shows the result of performing complex step differentiation using the same example as in Section 4.1.1. We can see from both exact and numerical solution that complex-step differentiation does not suffer from small values of ε, meaning that ε can be chosen arbitrarily small (Martins et al. 2001) as long as it does not reach the underflow threshold (Goldberg 1991). Notice that for large values of the stepsize ε complex step differentiation gives similar results than finite differences, while for small values of ε the performance of complex step differentiation is slightly worse than forward AD. This result emphasize the observation made in Section 3.9.3, complex step differentiation has many aspects in common with finite differences and AD based on dual numbers.

However, the difference between the methods also makes the complex step differentiation sometimes more efficient than both finite differences and AD (Lantoine et al. 2012), an effect that can be counterbalanced by the number of extra unnecessary operations that complex arithmetic requires (see last column in Figure 4) (Martins et al. 2003).

4.2 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

(i) How to handle matrix/Jacobian-vector products

(ii) Numerical stability of the forward/reverse solver

(iii) 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 solving forward ODE and computing their sensitivities at the same time. These include `CVODES` within `SUNDIALS` in C (Hindmarsh et al. 2005; Serban et al. 2005); `ODESSA` (Leis et al. 1988) and `FATODE` (discrete adjoints) (Zhang et al. 2014) both in Fortram; `SciMLSensitivity.jl` in Julia (Rackauckas et al. 2020); `Dolfin-adjoint` based on the `FEniCS` Project (Farrell et al. 2013; Mitusch et al. 2019); `DENSERKS` in Fortram (Alexe et al. 2007); `DASPKADJOINT` (Cao et al. 2002); and `Diffrax` in Python (Kidger 2021).

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 Krylov mode. Another important consideration is that all these methods have subroutines to compute the JVPs and VJPs involved in the sensitivity and adjoint equations, respectively. This calculation is carried out by another sensitivity method, usually finite differences or AD, which plays a central role when analyzing the accuracy and stability of the adjoint method.

4.2.1 Forward sensitivity equation

For systems of equations with few number of parameters, this method is useful since the system of $n(p+1)$ equations composed by Equations (1) and (36) can be solved using the same precision for both solution and sensitivity numerical evaluation. Furthermore, it does not required saving the solution in memory. The following example illustrates how Equation (67) and the sensitivity equation can be solved simultaneously using the simple explicit Euler method $\clubsuit_5$:

```julia
ω = 0.2
p = [ω]
u0 = [0.0, 1.0]
tspan = [0.0, 10.0]

# Dynamics
function f(u, p, t)
    du₁ = u[2]
    du₂ = - p[1]^2 * u[1]
    return [du₁, du₂]
end

# Jacobian ∂f/∂p
function ∂f∂p(u, p, t)
    Jac = zeros(length(u), length(p))
    Jac[2,1] = -2*p[1]*u[1]
    return Jac
end

# Jacobian ∂f/∂u
function ∂f∂u(u, p, t)
    Jac = zeros(length(u), length(u))
    Jac[1,2] = 1
    Jac[2,1] = -p[1]^2
    return Jac
end

# Explicit Euler method
function sensitivityequation(u0, tspan, p, dt)
    u = u0
    sensitivity = zeros(length(u), length(p))
    for ti in tspan[1]:dt:tspan[2]
        sensitivity += dt * (∂f∂u(u, p, ti) * sensitivity + ∂f∂p(u, p, ti))
        u += dt * f(u, p, ti)
    end
    return u, sensitivity
end

u, s = sensitivityequation(u0, tspan , p, 0.001)
```

The simplicity of the sensitivity method makes it available in most software for sensitivity analysis. In Julia, the `ODEForwardSensitivityProblem` method implements continuous sensitivity analysis, which generates the JVP operations required as part of the forward sensitivity equations via `ForwardDiff.jl` (see Section 3.9.4) or finite differences. The same result can be achieved by:

```julia
using Zygote, SciMLSensitivity
```

```julia
function f!(du, u, p, t)
    du[1] = u[2]
    du[2] = - p[1]^2 * u[1]
end

prob = ODEForwardSensitivityProblem(f!, u0, tspan, p)
sol = solve(prob, Tsit5())
```

Notice that in the last example we needed to re-define the out-of-place function `f` to the in-place version `f!`.

For stiff systems of ODEs the use of the sensitivity equations can be computationally unfeasible (Kim et al. 2021). This is because stiff ODEs require the use of stable solvers with cubic cost with respect to the number of ODEs (Wanner et al. 1996), making the total complexity of the sensitivity method $\mathcal{O}(n^3 p^3)$. This complexity makes this method expensive for models with many ODEs and/or parameters unless the solver is able to further specialize on sparsity or properties of the linear solver (i.e. through Newton-Krylov methods).

4.2.1.1 Computing JVPs and VJPs inside the solver

All solver-based methods require the computation of JVPs or VJPs. In the case of the sensitivity equation, this correspond to the JVPs resulting form the product $\frac{\partial f}{\partial u} s$ in Equation (36). For the adjoint equations, although an efficient trick has been used to remove the computationally expensive VJP, we still need to evaluate the term $\lambda^T \frac{\partial G}{\partial \theta}$ for the discrete adjoint method in Equation (43), and $\lambda^T \frac{\partial f}{\partial \theta}$ for the continuous adjoint method in Equation (57). Therefore, the choice of the algorithm to compute JVPS/VJPs can have a significant impact in the overall performance.

In `SUNDIALS`, the JVPS/VJPs involved in the sensitivity and adjoint method are handled using finite differences unless specified by the user (Hindmarsh et al. 2005). In `FATODE`, they can be computed with finite differences, AD, or it can be provided by the user. In the Julia ecosystem, different AD packages are available for this task (see Section 4.1.2.2), including `ForwardDiff.jl`, `ReverseDiff.jl`, `Zygote.jl` (Innes et al. 2019), `Enzyme.jl` (Moses et al. 2020), `Tracker.jl`. These will compute the JVPs/VJPs using some form of AD, which will result in correct calculations but potentially sub-optimal code. In Julia, the options `autodiff` and `autojacvec` allow one to customize if JVPs/VJPs are computed using AD or finite differences.

4.2.2 Adjoint methods

For complex and large systems (e.g. of more than 50 parameters and ODEs), direct methods for computing the gradient on top of the numerical solver can be memory expensive due to the large number of function evaluations required by the solver and the later store of the intermediate states. For these cases, adjoint-based methods allows us to compute the gradients of a loss function by instead computing the adjoint that serves as a bridge between the solution of the ODE and the final gradient. If done well the adjoint method offers considerate advantages when working with complex systems. Since we are dealing with a new differential equation special care needs to be taken with respect to numerical efficiency and stability.

4.2.2.1 Discrete adjoint method

In order to illustrate how the discrete adjoint method works, the following example shows how to manually solve for the gradient of the solution of (67) using an explicit Euler method.

```
function discrete_adjoint_method(u0, tspan, p, dt)
    u = u0
    times = tspan[1]:dt:tspan[2]

    λ = [1.0, 0.0]
    ∂L∂θ = zeros(length(p))
    u_store = [u]

    # Forward pass to compute solution
    for t in times[1:end-1]
        u += dt * f(u, p, t)
        push!(u_store, u)
    end

    # Reverse pass to compute adjoint
    for (i, t) in enumerate(reverse(times)[2:end])
        u_memory = u_store[end-i+1]
        λ += dt * ∂f∂u(u_memory, p, t)' * λ
        ∂L∂θ += dt * λ' * ∂f∂p(u_memory, p, t)
    end

    return ∂L∂θ
end

∂L∂θ = discrete_adjoint_method(u0, tspan, p, 0.001)
```

In this case, the full solution in the forward pass is stored in memory and used to compute the adjoint and integrate the loss function during the reverse pass. As in the case of reverse AD, checkpointing can be used here.

The previous example shows a manual implementation of the adjoint method. However, as we discuss in Section 3.9.3, the discrete adjoint method can be directly implemented using reverse AD. In the Julia SciML ecosystem, `ReverseDiffAdjoint` performs reverse AD on the numerical solver via `ReverseDiff.jl`; `ZygoteAdjoint` via `Zygote.jl`; and `TrackerAdjoint` via `Tracker.jl`.

4.2.2.2 Continuous adjoint method

The continuous adjoint methods offers a series of advantages over the discrete method and the rest of the forward methods previously discussed. Just as the discrete adjoint methods and backpropagation, the bottleneck is how to solve for the adjoint $\lambda(t)$ due to its dependency with VJPs involving the state $u(t)$. Effectively, notice that Equation (55) involves the terms $f(u, \theta, t)$ and $\frac{\partial h}{\partial u}$, which are both functions of $u(t)$. In opposition to the discrete adjoint methods, notice that here the full continuous trajectory $u(t)$ is needed, instead of its discrete pointwise evaluation. There are two solutions for addressing the evaluation of $u(t)$ during the backwards step.

(i) **Interpolation.** During the forward model, we can store in memory intermediate states of the numerical solution allowing the dense evaluation of the numerical solution at any given time. This can be done using dense output formulas, for example by adding extra stages to the Runge-Kutta scheme (Equation (2)) that allows to define a continuous interpolation, a method known as continuous Runge-Kutta (Alexe et al. 2009; Wanner et al. 1996).

(ii) **Backsolve.** Solve again the original system of ODEs together with the adjoint as the solution

	Method	Stability	Non-Stiff Performance	Stiff Performance	Memory
Discrete	ReverseDiffAdjoint	Good		$\mathcal{O}(n^3 + p)$	High
	ZygoteAdjoint	Good		$\mathcal{O}(n^3 + p)$	High
	TrackerAdjoint	Good		$\mathcal{O}(n^3 + p)$	High
Continuous	Sensitivity equation	Good		$\mathcal{O}(n^3 p^3)$	$\mathcal{O}(1)$
	Backsolve adjoint	Poor		$\mathcal{O}((n + p)^3)$	$\mathcal{O}(1)$
	Backsolve adjoint◀	Medium		$\mathcal{O}((n + p)^3)$	$\mathcal{O}(K)$
	Interpolating adjoint	Good		$\mathcal{O}((n + p)^3)$	High
	Interpolating adjoint◀	Good		$\mathcal{O}((n + p)^3)$	$\mathcal{O}(K)$
	Quadrature adjoint	Good		$\mathcal{O}(n^3 + p)$	High
	Gauss adjoint	Good		$\mathcal{O}(n^3 + p)$	High
	Gauss adjoint◀	Good		$\mathcal{O}(n^3 + p)$	$\mathcal{O}(K)$

Table 1: *Comparison in performance and cost of solver-based methods. Methods that can be checkpointed are indicated with the symbol ◀, with K the total number of checkpoints. The nomenclature of the different adjoint methods here follows the naming in the documentation of* `SciMLSensitivity.jl` *(Rackauckas et al. 2020).*

of the reversed augmented system (Chen et al. 2018)

$$\frac{d}{dt}\begin{bmatrix} u \\ \lambda \\ \frac{dL}{d\theta} \end{bmatrix} = \begin{bmatrix} -f \\ -\frac{\partial f}{\partial u}^T \lambda - \frac{\partial h}{\partial u}^T \\ -\lambda^T \frac{\partial f}{\partial \theta} - \frac{\partial h}{\partial \theta} \end{bmatrix} \qquad \begin{bmatrix} u \\ \lambda \\ \frac{dL}{d\theta} \end{bmatrix}(t_1) = \begin{bmatrix} u(t_1) \\ \frac{\partial L}{\partial u(t_1)} \\ \lambda(t_0)^T s(t_0) \end{bmatrix}. \tag{71}$$

An important problem with this approach is that computing the ODE backwards $\frac{du}{dt} = -f(u, \theta, t)$ can be unstable and lead to large numerical errors (Kim et al. 2021; Zhuang et al. 2020). 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. In practice, this method is hardly stable for most complex (even non-stiff) differential equations.

When dealing with stiff differential equations, special considerations need to be taken into account. Two alternatives are proposed in (Kim et al. 2021), 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 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 $\mathcal{O}(n^3 + p)$.

4.2.2.3 Continuous checkpointing

Both interpolating and backsolve adjoint methods can be implemented along with a checkpointing scheme. This can be done by choosing save points in the forward pass. For the interpolating methods, the interpolation is reconstructed in the backwards pass between two save points. This reduces the total memory requirement of the interpolating method to simply the maximum cost of holding an interpolation between two save points, but requires a total additional computational effort equal to one additional forward pass. In the backsolve variation, the value u in the reverse pass can be corrected to be the saved point, thus resetting the numerical error introduced during the backwards evaluation and thus improving the accuracy.

4.2.2.4 Solving the quadrature

Another computational consideration is how the integral in Equation (57) is numerically evaluated. While one can solve the integral simultaneously to the other equations using an ODE solver, this is only recommended with explicit methods as with implicit methods these additional ODEs are of size p and thus can increase the complexity of an implicit solve by $O(p^3)$. The interpolating adjoint and backsolve adjoint method use this ODE solver approach for computing the integrand. On the other side, the quadrature adjoint approach avoids this computational cost by computing the dense solution $\lambda(t)$ and then computing the quadrature

$$\begin{aligned}\frac{dL}{d\theta} &= \lambda(t_0)^T s(t_0) + \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta} \right) dt \\ &\approx \lambda(t_0)^T s(t_0) + \sum_{j=1}^{N} \omega_j \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta} \right) (\tau_i) \end{aligned} \tag{72}$$

where ω_i, τ_i are the weights and knots of the adaptive Gauss-Kronrod method from `QuadGK.jl` (Gonnet 2012; Laurie 1997). This method results in global error control on the integration and removes the cubic scaling within implicit solvers. Nonetheless, it requires a larger memory cost by storing the adjoint pass continuous solution.

Solvers designed for large implicit systems allow for solving explicit integrals based on the ODE solution simultaneously without including the equations in the ODE evaluation in order to avoid this expense. The Sundials COVDE solver introduced this technique specifically for BDF methods (Hindmarsh et al. 2005). In the Julia `DifferentialEquations.jl` solvers, this can be done using a callback (specifically the numerical integration callbacks form the `DiffEqCallbacks.jl` library). The Gauss adjoint method uses the callback approach to allow for a simultaneous explicit evaluation the integral using Gaussian quadrature related to (Norcliffe et al. 2023) but using a different approximation to improve convergence.

These differences in the schemes for computing $u(t)$ and the final quadrature give rise to the set of methods in Table 1. Excluding the sensitivity equation which was added for completeness, all of these are *the adjoint method* with differences being in the way steps of the adjoint method are approximated, and notably there is a general trade-off in stability, performance, and memory. One final piece to note is that while the Gauss adjoint achieves good properties according to this chart, the quadrature adjoint notably uses a global error control of the quadrature as opposed to the local error control of the Gauss adjoint, and thus can achieve more robust bounding of the error with respect to user chosen tolerances.

5 Recommendations

5.1 General guidelines

There is no sensitivity method that is universally suitable for all problems equally and that performs better under all conditions. However, in light of the methods we explore in this work, we can give

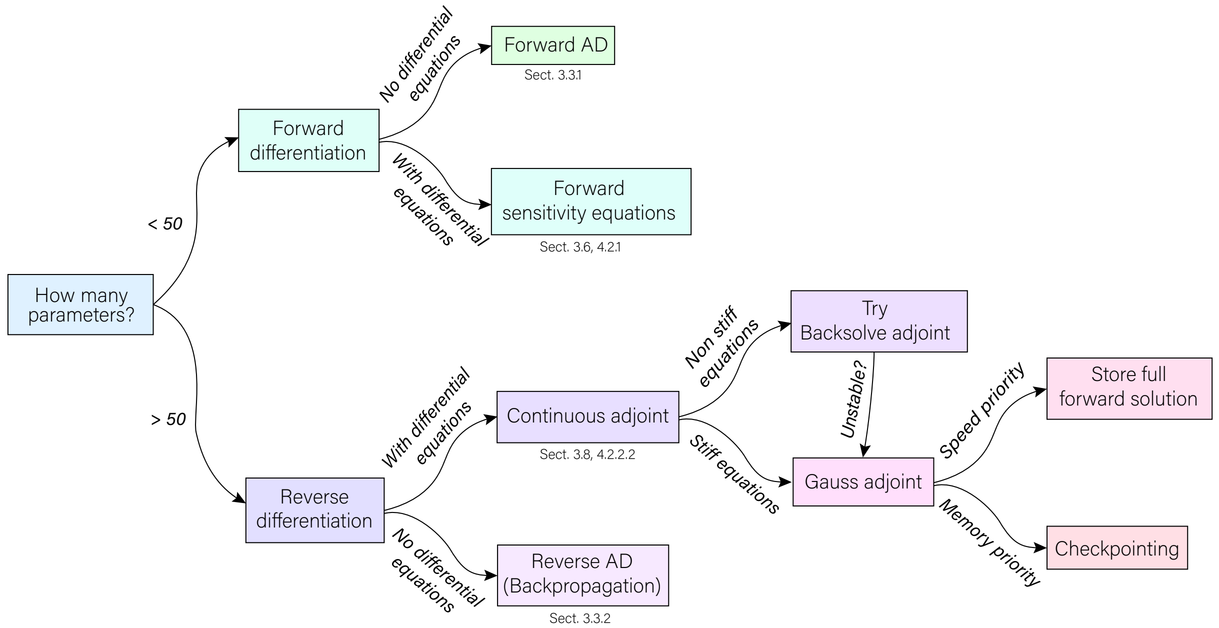

Figure 7: *Decision-making tree summarizing the choice of sensitivity methods for different problems depending the type of differential equation, number of parameters, and memory-time trade-off.*

general guidelines on which methods to use in specific circumstances. In this section we provide a practical overview of which methods are the most suitable for different situations. A simplified overview of this decision-making process is depicted in Figure 7.

Working with small systems

For sufficiently small systems of less than 50 parameters and ODEs, it has been shown that forward AD and sensitivity equations are the most efficient method, outperforming sensitivity and adjoint methods. The original benchmark of these methods is included in (Ma et al. 2021), though the `SciMLBenchmarks` system continually updates the benchmarks and has revised the cutoff point as reverse-mode AD engines improved. See `https://docs.sciml.ai/SciMLBenchmarksOutput/stable/` for continued updates. Furthermore, as we have shown in Section 4.1, automatic differentiation outperforms other forms of direct differentiation (finite differences, complex-step differentiation). Modern scientific software commonly support automatic differentiation, making forward AD our preferred choice for small problems. We will make further comments in the choice between discrete and continuous methods later in this section.

Working with large systems

For larger systems with more than 50 parameters and ODEs, reverse techniques are required. As explained in section 4.2, the continuous adjoint method, particularly the Gauss adjoint, and for very specific cases, the interpolating adjoint and quadrature adjoint, are the most suitable methods to tackle large stiff systems. The choice between these three types of adjoints will be problem-specific and will depend on the trade-off between numerical stability, performance and memory usage.

Adjoint methods supporting checkpointing present more flexibility in this sense, and can allow modulating the method depending on the performance vs memory constraints of each problem.

Unlike for small systems of ODEs and a reduced number of parameters, differentiating large ODE systems (e.g. stiff discretized PDEs) with respect to a large number of parameters (e.g. in a neural network), is a much more complex problem. Current state-of-the-art tools can easily work for a wide array of small systems (Rackauckas et al. 2020), whereas methods for large systems are still under heavy development and are likely to see many changes and improvements in the future.

Finally, it is worth mentioning that for complex stiff problems, following different strategies to avoid local minima will be absolutely necessary in order to achieve convergence (.e.g. using multiple shooting (Boussange et al. 2024; Kiehl et al. 2006) or progressively increasing the time span which is fitted).

Efficiency vs stability

When using discrete methods, it is important to be aware that the differentiation machinery is applied after the numerical solver for the differential equation has been specified, meaning that the derivatives are computed with respect to the time discretization instead of the solution (Eberhard et al. 1996). As we explored in Section 4.1.2.1, this can mean the method is non-convergent in the case where the iterative solver has adaptive stepsize controllers that depend on the parameter to differentiate. Although some solutions have been proposed to solve this in the case of discrete methods (Eberhard et al. 1996) (and have been implemented in Julia), this is a problem that continuous methods do not have since they apply the differentiation step before the numerical algorithm has been specified. Using many of the aforementioned *tricks*, such as continuous checkpointing and Gaussian quadrature approximations, continuous sensitivity analysis tends to be more memory and computationally efficient. However, the errors of discrete adjoint method's derivative error may better represent the actual code being evaluated, and thus in some instances has been found to lead to more stable optimizations.

In a nutshell, continuous adjoint methods tend to be more efficient while discrete adjoint methods tend to be more stable (Rackauckas et al. 2020), though the opposite can apply and as such the choice ultimately depends on the nuances of each problem.

Choosing a direct method

When computing the gradient of a generic function other than a numerical solver, we further recommend the use of automatic differentiation (reverse or forward depending the number of parameters) as the direct method of choice, outperforming finite differences, complex step differentiation, and symbolic differentiation. This recommendation applies also for the inner VJPs and JVPs calculations performed inside the numerical solver (Section 4.2.1.1). As we had seen in Section 4.1, finite differences and complex step differentiation do not really provide an advantage over AD in terms of precision and require the tuning of the stepsize ε. On the other hand, if symbolic differentiation can be more efficient in nested cases (see ...) or when the spartity patter of the Jacobian is known, in general this advantage is not drastic in most real cases and can generate difficulties when used inside the numerical solver.

However, it is important to remark that this recommendation is constraint to the availability and interoperability of different AD and sensitivity software. For example, when computing higher-order derivatives multiple layers of direct methods became more difficult to implement and may result in complicated computer programs. In this cases, complex step differentiation may offer a simple alternative with similar performance than AD for small stepsizes.

Taking into account model architecture

While it is important to take into account the mathematical aspects of the problem in order to choose the right type of sensitivity method, in practice, code structure and characteristics have a very strong impact in the choice of which packages to use to compute the sensitivities. In the context of the Julia and Python, different packages using different AD techniques will be able to deal with certain situations, while others will fail. Some of the most commonly encountered current limitations are the following ones:

- The use of branching (i.e. `if/else` statements) and control flow (i.e. `for` and `while` loops) presents issues for tape-based AD methods. This is currently not supported by `ReverseDiff.jl` and partially supported by `JAX` in Python. Non-tape-based AD methods tend to support this, like `Enzyme.jl` and `Zygote.jl`.
- Mutation of arrays (i.e. in-place operations) is sometimes problematic, since it does not allow to preserve the chain rule during reverse differentation for some packages like `Zygote.jl` or `JAX`, while it is currently supported by `ReverseDiff.jl` and `Enzyme.jl`.
- Compatibility with GPUs (Graphical Processing Units) is still greatly under development for sensitivity methods. Some AD packages like `JAX`, `Enzyme.jl` and `Zygote.jl` support it, while other packages like `ReversDiff.jl` do not.

It is important to bear in mind that direct methods are easier to implement in programming languages where AD already exists and sometimes does not required any special package, like for the Julia programming language. Nonetheless, users must be aware of the aforementioned convergence issues of AD naively applied to solvers. Thus we recommend the use of robust and tested software when available (e.g. the Julia SciML ecosystem or Diffrax in Python) as the solvers must apply corrections to AD implementations in order to guarantee numerically correct derivatives.

5.2 Generalization to PDEs and DAEs

5.3 Chaotic systems

All the sensitivity methods discussed in the previous sections encounter challenges and become less useful when applied to chaotic systems (Wang et al. 2014a). To illustrate this, let us consider long-time-averaged quantities

$$\langle L(\theta) \rangle_T = \frac{1}{T} \int_0^T L(u(t), \theta)\, dt, \tag{73}$$

of chaotic systems, where $L(u(t), \theta)$ is the instantaneous objective and $u(t)$ denotes the state of the dynamical system at time t. For ergodic dynamical systems, $\lim_{T\to\infty} \langle L(\theta) \rangle_T$ depends solely on the governing dynamical system and is independent of the specific choice of trajectory $u(t)$. In particular, $\lim_{T\to\infty} \langle L(\theta) \rangle_T$ does not depend on the initial condition. Under the assumption of uniform hyperbolic systems, it is possible to derive closed-form expressions and differentiability conditions for $\langle L(\theta) \rangle_T$ (Ruelle 1997, 2009). However, computing derivatives using numerical methods of statistical quantities of the form (73) with respect to the vector parameter θ in chaotic dynamical systems remains challenging due to the *butterfly effect*, i.e. small changes in the initial state or parameter can result in large differences in a later state (**Lorenz.1963**). As a consequence, the solutions of the forward and adjoint sensitivity equations blow up (exponentially fast) instead of converging to the actual derivative. To address these issues, various modifications and methods have been proposed, including approaches based on ensemble averages (Eyink et al. 2004; Lea et al. 2000b), the Fokker-Planck equation (Blonigan et al. 2014; Thuburn 2005), the fluctuation-dissipation theorem (Abramov et al. 2007, 2008; Leith 1975), shadowing lemma (Blonigan 2017; Blonigan et al. 2018; Ni et al. 2019a, 2017, 2019b; Wang 2013, 2014; Wang et al. 2014b), and modifications of Ruelle's formula (Chandramoorthy et al. 2022; Ni 2020).

In Julia, this is implemented in the sensitivity method `AdjointLSS` and `NILSS`. Standard derivative approximations are inappropriate for such systems and will not give convergent estimates. This includes AD of a solver.

6 Conclusions

In the present work, we presented a comprehensive overview of the different existing methods for calculating the sensitivity or gradients of forward maps involving numerical solutions of differential equations. This task has been approached from three different angles. First, we presented the existing literature in different scientific communities where adjoints and sensitivities have been used before and play a central modelling role, especially for inverse modeling. Secondly, we reviewed the mathematical foundations of these methods and their classification as forward-reverse and discrete-continuous. Then, we have shown how the different methods are implemented in the Julia programming language and the different computational considerations that we must take into account when implementing or using a sensitivity algorithm. There exist a myriad of options and combinations to compute sensitivities of functions involving differential equations, further complicated by jargon and scientific cultures of different communities. We believe this review paper provides a clearer overview on this topic, and can serve as an entry point to navigate this field and help in terms of decision-making of the most suitable methods for many different scientific problems.

Appendices

A Lagrangian formulation of the adjoint method

The adjoint equation can be derived directly from the Following the analysis in (Giles et al. 2000b), we decided to present both approaches here, although we prefer with the duality viewpoint introduced in the main text since we believe is more commonly used and easy to understand for newcomers.

In this section we are going to derive the adjoint equation for both discrete and continuous methods using the Lagrangian formulation of the adjoint. It is important to mention that this is different than using the Lagrange multipliers approach, a common confusion in the literature(Givoli 2021). 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.

A.1 Discrete adjoint

Following the notation introduced in Section 3.7, we first define the Lagrangian function $I(U,\theta)$ as

$$I(U,\theta) = L(u;\theta) + \lambda^T G(U;\theta), \tag{74}$$

where $\lambda \in \mathbb{R}^{nk}$ is, in principle, any choice of vector. Given that for every choice of the parameter θ we have $G(U;\theta) = 0$, we have that $I(U,\theta) = L(U,\theta)$. Then, the gradient of L with respect to the vector parameter θ can be computed as

$$\begin{aligned}\frac{dL}{d\theta} = \frac{dI}{d\theta} &= \frac{\partial L}{\partial\theta} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial\theta} + \lambda^T\left(\frac{\partial G}{\partial U} + \frac{\partial G}{\partial U}\frac{\partial U}{\partial\theta}\right) \\ &= \frac{\partial L}{\partial\theta} + \lambda^T\frac{\partial G}{\partial U} + \left(\frac{\partial L}{\partial\theta} + \lambda^T\frac{\partial G}{\partial U}\right)\frac{\partial U}{\partial\theta}.\end{aligned} \tag{75}$$

The important trick in the last term involved grouping all the terms involving the sensitivity $\frac{\partial U}{\partial\theta}$ together. In order to avoid the computation of the sensitivity at all, we can define λ as the vector that makes the last term in the right hand side of Equation (75) which results in the same results we obtained in Equation (42) and (43), that is,

$$\frac{\partial L}{\partial\theta} = -\lambda^T\frac{\partial G}{\partial U}. \tag{76}$$

Finally, the gradient can be computed as

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial\theta} + \lambda^T\frac{\partial G}{\partial U}. \tag{77}$$

A.2 Continuous adjoint

For the continuous adjoint method, we proceed the same way by defining the Lagrangian $I(\theta)$ as

$$I(\theta) = L(\theta) - \int_{t_0}^{t_1} \lambda(t)^T\left(\frac{du}{dt} - f(u,\theta,t)\right)dt \tag{78}$$

where $\lambda(t) \in \mathbb{R}^n$ is any function. It is important to mention that, in principle, there is connection yet between $\lambda(t)$ and the Lagrange multiplier associated to the constraint (Givoli 2021). Instead, the condition

$$\int_{t_0}^{t_1} \lambda(t)^T\left(\frac{du}{dt} - f(u,\theta,t)\right)dt = 0 \qquad \text{for all function } \lambda : [t_0,t_1] \mapsto \mathbb{R}^n \tag{79}$$

correspond to the weak formulation of the differential equation (1) (Brézis 2011). As long as the differential equation is satisfyed, we have $I(\theta) = L(\theta)$ for all choices of the vector parameter θ. Now,

$$\frac{dL}{d\theta} = \frac{dI}{d\theta} = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(\frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u}\frac{\partial u}{\partial \theta} \right) dt - \int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \left(\frac{d}{dt}\frac{du}{d\theta} - \frac{\partial f}{\partial u}\frac{du}{d\theta} - \frac{\partial f}{\partial \theta} \right) dt. \tag{80}$$

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

$$\begin{aligned} \frac{dL}{d\theta} = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta} \right) dt - \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(-\frac{d\lambda^T}{dt} - \lambda^T \frac{\partial f}{\partial u} - \frac{\partial h}{\partial u} \right) s(t)\, dt \\ - \Big(\lambda(t_1)^T s(t_1) - \lambda(t_0)^T s(t_0) \Big). \end{aligned} \tag{81}$$

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

$$\frac{d\lambda}{d\theta} = - \left(\frac{\partial f}{\partial u} \right)^T \lambda - \left(\frac{\partial h}{\partial u} \right)^T, \tag{82}$$

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.

A.3 The adjoint from a functional analysis perspective

B When AD is algorithmically correct but numerically wrong

In this section, we are going to consider certain errors that can potentially show up when combining AD with a numerical solver. Numerical solvers for differential equations usually estimate internally a scaled error computed as define an error term computed as (Hairer et al. 2008; Rackauckas et al. 2016)

$$\text{Err}^{n+1}_{\text{scaled}} = \left(\frac{1}{n} \sum_{i=1}^{n} \left(\frac{\text{err}_i^{n+1}}{\mathfrak{abstol} + \mathfrak{reltol} \times M} \right)^2 \right)^{\frac{1}{2}}, \tag{83}$$

with $\mathfrak{abstol}$ and $\mathfrak{reltol}$ the absolute and relative solver tolerances (customize), respectively; M is the maximum expected value of the numerical solution; and err_i^{n+1} is an estimation of the numerical error at step $n+1$. Common choices for these include $M = \max(u_i^{n+1}, \hat{u}_i^{n+1})$ and $\text{err}_i^{n+1} = u_i^{n+1} - \hat{u}_i^{n+1}$, but these can vary between solvers. Estimations of the local error err_i^{n+1} can be based on two approximation to the solution based on Runge-Kutta pairs (Hairer et al. 2008; Ranocha et al. 2022); or in theoretical upper bounds provided by the numerical solver. The choice of the norm $\frac{1}{\sqrt{n}} \| \cdot \|_2$ for computing the total error $\text{Err}_{\text{scaled}}$, sometimes known as Hairer norm, has been the standard for a long time(Ranocha et al. 2022) and it is based on the assumption that a mere increase in the size of the systems of ODEs (e.g., by simply duplicating the ODE system) should not affect the stepsize choice, but other options can be considered (Hairer et al. 2008).

The goal of a stepize controller is to pick Δt_{n+1} as large as possible (so the solver requires less total steps) at the same time that $\text{Err}_{\text{scaled}} \leq 1$. One of the most used methods to archive this is the proportional-integral controller (PIC) that updates the stepsize according to (Ranocha et al. 2022; Wanner et al. 1996)

$$\Delta t_{n+1} = \eta \, \Delta t_n \qquad \eta = w_{n+1}^{\beta_1/q} w_n^{\beta_2/q} w_{n-1}^{\beta_3/q} \tag{84}$$

with $w_{n+1} = 1/\text{Err}^{n+1}_{\text{scaled}}$ the inverse of the scaled error estimates; β_1, β_2, and β_3 numerical coefficients defined by the controller; and q the order of the numerical solver. If the stepsize Δt_{n+1} proposed in Equation (84) to update from u^{n+1} to u^{n+2} does not satisfy $\text{Err}^{n+2}_{\text{scaled}} \leq 1$, a new smaller stepsize is proposed. When $\eta < 1$ (which is the case for simple controllers with $\beta_2 = \beta_3 = 0$), Equation (84) can be used for the local update. It is also common to restrict $\eta \in [\eta_{\min}, \eta_{\max}]$ so the stepsize does not change abruptly (Hairer et al. 2008).

When performing forward AD though numerical solver, the error used in the stepsize controller needs to naturally account for both the errors induced in the numerical solution of the original ODE and the errors in the dual component carrying the value of the sensitivity. This relation between the numerical solver and AD has been made explicit when we presented the relationship between forward AD and the sensitivity equations (Section 3.6, Equation (63)). To illustrate this, let us consider the simple ODE example from Section 4.1.2.1, consisting in the system of equations

$$\begin{cases} \frac{du_1}{dt} = a u_1 - u_1 u_2 & u_1(0) = 1 \\ \frac{du_2}{dt} = -a u_2 + u_1 u_2 & u_2(0) = 1. \end{cases} \tag{85}$$

Notice that for $a = 1$ this ODE admits a simple analytical solution $u_1(t) = u_2(t) = 1$ for all times t, making this problem very simple to solve numerically. The following code solves for the derivative with respect to the parameter a using two different methods[1]. The second method using forward AD with dual numbers declares the `internalnorm` argument according to Equation (83).

[1]Full code available at ...

```
using SciMLSensitivity, OrdinaryDiffEq, Zygote, ForwardDiff

function fiip(du, u, p, t)
    du[1] =  p[1] * u[1] - u[1] * u[2]
    du[2] = -p[1] * u[2] + u[1] * u[2]
end

p = [1.]
u0 = [1.0;1.0]
prob = ODEProblem(fiip, u0, (0.0, 10.0), p);

# Correct gradient computed using
grad0 = Zygote.gradient(p->sum(solve(prob, Tsit5(), u0=u0, p=p, sensealg =
    ForwardSensitivity(), saveat = 0.1, abstol=1e-12, reltol=1e-12)), p)
# grad0 = ([212.71042521681443],)

# Original AD with wrong norm
grad1 = Zygote.gradient(p->sum(solve(prob, Tsit5(), u0=u0, p=p, sensealg =
    ForwardDiffSensitivity(), saveat = 0.1, internalnorm = (u,t) -> sum(abs2,u/
    length(u)), abstol=1e-12, reltol=1e-12)), p)
# grad1 = ([6278.15677493293],)
```

The reason why the two methods give different answers is because the error estimation by the stepsize controller is ignoring numerical errors in the dual component. In the later case, the local estimated error is drastically underestimated to $\text{err}_i^{n+1} = 0$, which makes the stepsize Δt_{n+1} to increase by a multiplicative factor at evary step. This can be fixed by instead considering a norm that accounts for both the primal and dual components in the forward pass,

$$\text{Err}_{\text{scaled}}^{n+1} = \left[\frac{1}{n(p+1)} \left(\sum_{i=1}^{n} \left(\frac{u_i^{n+1} - \hat{u}_i^{n+1}}{\mathfrak{abstol} + \mathfrak{reltol} \max(u_i^{n_1}, \hat{u}_i^{n+1})} \right)^2 + \sum_{i=1}^{n} \sum_{j=1}^{p} \left(\frac{s_{ij}^{n+1} - \hat{s}_{ij}^{n+1}}{\mathfrak{abstol} + \mathfrak{reltol} \max(s_{ij}^{n_1}, \hat{s}_{ij}^{n+1})} \right)^2 \right) \right]^{\frac{1}{2}}, \tag{86}$$

which now gives the right answer

```
sse(x::Number) = x^2
sse(x::ForwardDiff.Dual) = sse(ForwardDiff.value(x)) + sum(sse, ForwardDiff.
    partials(x))

totallength(x::Number) = 1
totallength(x::ForwardDiff.Dual) = totallength(ForwardDiff.value(x)) + sum(
    totallength, ForwardDiff.partials(x))
totallength(x::AbstractArray) = sum(totallength,x)

grad3 = Zygote.gradient(p->sum(solve(prob, Tsit5(), u0=u0, p=p, sensealg =
    ForwardDiffSensitivity(), saveat = 0.1, internalnorm = (u,t) -> sqrt(sum(x-
    >sse(x),u) / totallength(u)), abstol=abstol, reltol=reltol)), p)
# grad3 = ([212.71042521681392],)
```

Notice that current implementations of forward AD inside `SciMLSensitivity.jl` already accounts for this and there is no need to specify the internal norm ♣3.

C Supplementary code

This is a list of the code provided along with the current manuscript. All the following scripts can be found in the GitHub repository `DiffEqSensitivity-Review`.

♣$_1$ **Comparison of direct methods.** The script `direct-comparision.jl` reproduces Figure 6.

♣$_2$ **Dual numbers definition.** The script `dualnumber_definition.jl` includes a very simple example of how to define a dual number using `struct` in Julia and how to extend simple unary and binary operations to implement the chain rule usign multiple distpatch.

♣$_3$ **When AD is algorithmically correct but numerically wrong.** The script `example-AD-tolerances.` includes the example shown in Section 4.1.2.1 and further elaborated in Appendix B where forward AD gives the wrong answer when tolerances in the gradient are not computed taking into account both numerical errors in the numerical solution and the sensitivity matrix. Further examples of this phenomena can be found in the Python script `testgradient_python.py` and the Julia `testgradient_julia.jl`.

♣$_4$ **Complex step in numerical solver.** The script `complex_solver.jl` shows how to define the dynamics of the ODE to support complex variables and then compute the complex step derivative.

♣$_5$ **Solving the sensitivity equation.** The scrip `sensitivityequation.jl` includes a manual implementation of the sensitivity equations. This also includes how to compute the same sensitivity using `ForwardSensitivity` in Julia.

References

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng (2016). "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, pp. 265–283.

Abarbanel, H. D. I., P. J. Rozdeba, and S. Shirman (Aug. 2018). "Machine Learning: Deepest Learning as Statistical Data Assimilation Problems". In: *Neural Computation* 30.8, pp. 2025–2055. DOI: 10.1162/neco_a_01094.

Abdelhafez, M., B. Baker, A. Gyenis, P. Mundada, A. A. Houck, D. Schuster, and J. Koch (2020). "Universal gates for protected superconducting qubits using optimal control". In: *Physical Review A* 101.2, p. 022321. DOI: 10.1103/PhysRevA.101.022321.

Abdelhafez, M., D. I. Schuster, and J. Koch (2019). "Gradient-based optimal control of open quantum systems using quantum trajectories and automatic differentiation". In: *Phys. Rev. A* 99.5, p. 052327. DOI: 10.1103/PhysRevA.99.052327.

Abramov, R. V. and A. J. Majda (2007). "Blended response algorithms for linear fluctuation-dissipation for complex nonlinear dynamical systems". In: *Nonlinearity* 20.12, p. 2793. DOI: https://iopscience.iop.org/article/10.1088/0951-7715/20/12/004.

— (2008). "New approximations and tests of linear fluctuation-response for chaotic nonlinear forced-dissipative dynamical systems". In: *Journal of Nonlinear Science* 18, pp. 303–341. DOI: https://doi.org/10.1007/s00332-007-9011-9.

Aide, T. M., C. Corrada-Bravo, M. Campos-Cerqueira, C. Milan, G. Vega, and R. Alvarez (July 16, 2013). "Real-Time Bioacoustics Monitoring and Automated Species Identification". In: *PeerJ* 1.1, e103. DOI: 10.7717/peerj.103.

Åkesson, A., A. Curtsdotter, A. Eklöf, B. Ebenman, J. Norberg, and G. Barabás (Dec. 6, 2021). "The Importance of Species Interactions in Eco-Evolutionary Community Dynamics under Climate Change". In: *Nature Communications* 12.1, p. 4759. DOI: 10.1038/s41467-021-24977-x.

Alexe, M. and A. Sandu (2007). "DENSERKS: Fortran sensitivity solvers using continuous, explicit Runge-Kutta schemes". In.

— (2009). "Forward and adjoint sensitivity analysis with continuous explicit Runge–Kutta schemes". In: *Applied Mathematics and Computation* 208.2, pp. 328–346. DOI: 10.1016/j.amc.2008.11.035.

Allaire, G., C. Dapogny, and P. Frey (2014). "Shape optimization with a level set based mesh evolution method". In: *Computer Methods in Applied Mechanics and Engineering* 282, pp. 22–53.

Alsos, I. G., V. Boussange, D. P. Rijal, M. Beaulieu, A. G. Brown, U. Herzschuh, J.-C. Svenning, and L. Pellissier (Nov. 7, 2023). *Using Ancient Sedimentary DNA to Forecast Ecosystem Trajectories under Climate Change*. preprint. In Review. DOI: 10.21203/rs.3.rs-3542192/v1.

Arrazola, J. M., S. Jahangiri, A. Delgado, J. Ceroni, J. Izaac, A. Száva, U. Azad, R. A. Lang, Z. Niu, O. D. Matteo, R. Moyard, J. Soni, M. Schuld, R. A. Vargas-Hernández, T. Tamayo-Mendoza, C. Y.-Y. Lin, A. Aspuru-Guzik, and N. Killoran (2021). "Differentiable quantum computational chemistry with PennyLane". In: *arXiv*. DOI: 10.48550/arxiv.2111.09967.

Ascher, U. M. (2008). *Numerical methods for evolutionary differential equations*. SIAM.

Ascher, U. M. and C. Greif (2011). *A First Course in Numerical Methods*. Philadelphia, PA: Society for Industrial and Applied Mathematics. DOI: 10.1137/9780898719987.

Barnosky, A. D., E. A. Hadly, J. Bascompte, E. L. Berlow, J. H. Brown, M. Fortelius, W. M. Getz, J. Harte, A. Hastings, P. A. Marquet, N. D. Martinez, A. Mooers, P. Roopnarine, G. Vermeij, J. W. Williams, R. Gillespie, J. Kitzes, C. Marshall, N. Matzke, D. P. Mindell, E. Revilla, and A. B. Smith (2012). "Approaching a State Shift in Earth's Biosphere". In: *Nature* 486.7401, pp. 52–58. DOI: 10.1038/nature11018.

Barton, R. R. (1992). "Computing Forward Difference Derivatives In Engineering Optimization". In: *Engineering Optimization* 20.3, pp. 205–224. DOI: 10.1080/03052159208941281.

Bauer, F. L. (1974). "Computational Graphs and Rounding Error". In: *SIAM Journal on Numerical Analysis* 11.1, pp. 87–96. DOI: 10.1137/0711010.

Bauer, P., A. Thorpe, and G. Brunet (2015). "The quiet revolution of numerical weather prediction". In: *Nature* 525.7567, pp. 47–55. DOI: 10.1038/nature14956.

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (Jan. 2017). "Automatic Differentiation in Machine Learning: A Survey". In: *J. Mach. Learn. Res.* 18.1, pp. 5595–5637.

Bell, B. M. and J. V. Burke (2008). "Advances in Automatic Differentiation". In: *Lecture Notes in Computational Science and Engineering*, pp. 67–77. DOI: 10.1007/978-3-540-68942-3_7.

Bennett, C. H. (1973). "Logical Reversibility of Computation". In: *IBM Journal of Research and Development* 17.6, pp. 525–532. DOI: 10.1147/rd.176.0525.

Betancourt, M. (2017). "A Conceptual Introduction to Hamiltonian Monte Carlo". In: *arXiv*. DOI: 10.48550/arxiv.1701.02434.

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (2017). "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1, pp. 65–98. DOI: 10.1137/141000671.

Bezanson, J., S. Karpinski, V. B. Shah, and A. Edelman (2012). "Julia: A Fast Dynamic Language for Technical Computing". In: *arXiv*. DOI: 10.48550/arxiv.1209.5145.

Blessing, S., T. Kaminski, F. Lunkeit, I. Matei, R. Giering, A. Köhl, M. Scholze, P. Herrmann, K. Fraedrich, and D. Stammer (2014). "Testing variational estimation of process parameters and initial conditions of an earth system model". In: *Tellus A* 66.0, p. 22606. DOI: 10.3402/tellusa.v66.22606.

Blonigan, P. J. (2017). "Adjoint sensitivity analysis of chaotic dynamical systems with non-intrusive least squares shadowing". In: *Journal of Computational Physics* 348, pp. 803–826. DOI: https://doi.org/10.1016/j.jcp.2017.08.002.

Blonigan, P. J. and Q. Wang (2014). "Probability density adjoint for sensitivity analysis of the mean of chaos". In: *Journal of Computational Physics* 270, pp. 660–686. DOI: https://doi.org/10.1016/j.jcp.2014.04.027.

— (2018). "Multiple shooting shadowing for sensitivity analysis of chaotic dynamical systems". In: *Journal of Computational Physics* 354, pp. 447–475. DOI: https://doi.org/10.1016/j.jcp.2017.10.032.

Bocquet, M., J. Brajard, A. Carrassi, and L. Bertino (July 10, 2019). "Data Assimilation as a Learning Tool to Infer Ordinary Differential Equation Representations of Dynamical Models". In: *Nonlinear Processes in Geophysics* 26.3, pp. 143–162. DOI: 10.5194/npg-26-143-2019.

Bolibar, J., F. Sapienza, F. Maussion, R. Lguensat, B. Wouters, and F. Pérez (2023). "Universal differential equations for glacier ice flow modelling". In: *Geoscientific Model Development* 16.22, pp. 6671–6687. DOI: 10.5194/gmd-16-6671-2023.

Boussange, V., P. V. Aceituno, F. Schäfer, and L. Pellissier (2024). "Partitioning time series to improve process-based models with machine learning". In: *bioRxiv*. DOI: 10.1101/2022.07.25.501365.

Boussange, V., S. Becker, A. Jentzen, B. Kuckuck, and L. Pellissier (Dec. 1, 2023). “Deep Learning Approximations for Non-Local Nonlinear PDEs with Neumann Boundary Conditions”. In: *Partial Differential Equations and Applications* 4.6, p. 51. DOI: 10.1007/s42985-023-00244-0.

Boussange, V. and L. Pellissier (Dec. 6, 2022). “Eco-Evolutionary Model on Spatial Graphs Reveals How Habitat Structure Affects Phenotypic Differentiation”. In: *Communications Biology* 5.1, p. 668. DOI: 10.1038/s42003-022-03595-3.

Bradley, A. M. (2013). *PDE-constrained optimization and the adjoint method*. Tech. rep. Technical Report. Stanford University. https://cs. stanford. edu/
textasciitilde ambrad~...

Brajard, J., A. Carrassi, M. Bocquet, and L. Bertino (2021). “Combining Data Assimilation and Machine Learning to Infer Unresolved Scale Parametrization”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379.2194. DOI: 10.1098/rsta.2020.0086.

Brézis, H. (2011). *Functional analysis, Sobolev spaces and partial differential equations*. Vol. 2. 3. Springer.

Bryson, A., Y.-C. Ho, and G. Siouris (July 1979). “Applied Optimal Control: Optimization, Estimation, and Control”. In: *Systems, Man and Cybernetics, IEEE Transactions on* 9, pp. 366–367. DOI: 10.1109/TSMC.1979.4310229.

Bui-Thanh, T., C. Burstedde, O. Ghattas, J. Martin, G. Stadler, and L. C. Wilcox (2012). “Extreme-scale UQ for Bayesian inverse problems governed by PDEs”. In: *IEEE Computer Society Press*, p. 3.

Buizza, R. and T. N. Palmer (1995). “The singular-vector structure of the atmospheric global circulation”. In: *Journal of the Atmospheric Sciences* 52.9, pp. 1434–1456. DOI: 10.1175/1520-0469(1995)052<1434:tsvsot>2.0.co;2.

Butcher, J. C. (2001). “Numerical Analysis: Historical Developments in the 20th Century”. In: pp. 449–477. DOI: 10.1016/b978-0-444-50617-7.50018-5.

Butcher, J. and G. Wanner (1996). “Runge-Kutta methods: some historical notes”. In: *Applied Numerical Mathematics* 22.1–3, pp. 113–151. DOI: 10.1016/s0168-9274(96)00048-7.

Cao, Y., S. Li, and L. Petzold (2002). “Adjoint sensitivity analysis for differential-algebraic equations: algorithms and software”. In: *Journal of Computational and Applied Mathematics* 149.1, pp. 171–191. DOI: 10.1016/s0377-0427(02)00528-9.

Carrassi, A., M. Bocquet, L. Bertino, and G. Evensen (Sept. 9, 2018). “Data Assimilation in the Geosciences: An Overview of Methods, Issues, and Perspectives”. In: *WIREs Climate Change* 9.5, pp. 1–50. DOI: 10.1002/wcc.535.

Chalmandrier, L., F. Hartig, D. C. Laughlin, H. Lischke, M. Pichler, D. B. Stouffer, and L. Pellissier (Dec. 2021). “Linking Functional Traits and Demography to Model Species-Rich Communities”. In: *Nature Communications* 12.1, p. 2724. DOI: 10.1038/s41467-021-22630-1.

Chandramoorthy, N. and Q. Wang (2022). “Efficient computation of linear response of chaotic attractors with one-dimensional unstable manifolds”. In: *SIAM Journal on Applied Dynamical Systems* 21.2, pp. 735–781. DOI: https://doi.org/10.1137/21M1405599.

Chen, R. T., Y. Rubanova, J. Bettencourt, and D. K. Duvenaud (2018). “Neural ordinary differential equations”. In: *Advances in neural information processing systems* 31.

Christianson, B. (1994). “Reverse accumulation and attractive fixed points”. In: *Optimization Methods and Software* 3.4, pp. 311–326.

— (1998). “Reverse aumulation and imploicit functions”. In: *Optimization Methods and Software* 9.4, pp. 307–322.

Clifford (1871). "Preliminary sketch of biquaternions". In: *Proceedings of the London Mathematical Society* 1.1, pp. 381–395.

Courtier, P. and O. Talagrand (1987). "Variational Assimilation of Meteorological Observations With the Adjoint Vorticity Equation. Ii: Numerical Results". In: *Quarterly Journal of the Royal Meteorological Society* 113.478, pp. 1329–1347. DOI: 10.1002/qj.49711347813.

Coveney, P. V., E. R. Dougherty, and R. R. Highfield (2016). "Big data need big theory too". In: *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences* 374.2080, pp. 20160153–11. DOI: 10.1098/rsta.2016.0153.

Cox, D. R. and B. Efron (2017). "Statistical thinking for 21st century scientists". In: *Science Advances* 3.6, e1700768. DOI: 10.1126/sciadv.1700768.

Curtsdotter, A., H. T. Banks, J. E. Banks, M. Jonsson, T. Jonsson, A. N. Laubmeier, M. Traugott, and R. Bommarco (Feb. 7, 2019). "Ecosystem Function in Predator–Prey Food Webs—Confronting Dynamic Models with Empirical Data". In: *Journal of Animal Ecology* 88.2. Ed. by D. Stouffer, pp. 196–210. DOI: 10.1111/1365-2656.12892.

Dahlquist, G. (1985). "33 years of numerical instability, Part I". In: *BIT Numerical Mathematics* 25.1, pp. 188–204. DOI: 10.1007/bf01934997.

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

Der Houwen, P. J. van and B. P. Sommeijer (1980). "On the internal stability of explicit, m-stage Runge-Kutta methods for large m-values". In: *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik* 60.10, pp. 479–485.

Dimet, F.-X. L., I. M. Navon, and D. N. Daescu (2002). "Second-Order Information in Data Assimilation*". In: *Monthly Weather Review* 130.3, pp. 629–648. DOI: 10.1175/1520-0493(2002)130<0629:soiida>2.0.co;2.

Dorigo, T., A. Giammanco, P. Vischia, M. Aehle, M. Bawaj, A. Boldyrev, P. d. C. Manzano, D. Derkach, J. Donini, A. Edelen, F. Fanzago, N. R. Gauger, C. Glaser, A. G. Baydin, L. Heinrich, R. Keidel, J. Kieseler, C. Krause, M. Lagrange, M. Lamparth, L. Layer, G. Maier, F. Nardi, H. E. S. Pettersen, A. Ramos, F. Ratnikov, D. Röhrich, R. R. d. Austri, P. M. R. d. Árbol, O. Savchenko, N. Simpson, G. C. Strong, A. Taliercio, M. Tosi, A. Ustyuzhanin, and H. Zaraket (2022). "Toward the End-to-End Optimization of Particle Physics Instruments with Differentiable Programming: a White Paper". In: *arXiv*. DOI: 10.48550/arxiv.2203.13818.

Dormann, C. F. (Sept. 3, 2007). "Promising the Future? Global Change Projections of Species Distributions". In: *Basic and Applied Ecology* 8.5, pp. 387–397. DOI: 10.1016/j.baae.2006.11.001.

Dürrbaum, A., W. Klier, and H. Hahn (2002). "Comparison of Automatic and Symbolic Differentiation in Mathematical Modeling and Computer Simulation of Rigid-Body Systems". In: *Multibody System Dynamics* 7.4, pp. 331–355. DOI: 10.1023/a:1015523018029.

Dutkiewicz, S., M. J. Follows, P. Heimbach, and J. Marshall (2006). "Controls on ocean productivity and air-sea carbon flux: An adjoint model sensitivity study". In: *Geophysical Research Letters* 33.2, pp. 159–4. DOI: 10.1029/2005gl024987.

Eberhard, P. and C. Bischof (1996). "Automatic differentiation of numerical integration algorithms". In: *Mathematics of Computation* 68.226, pp. 717–731. DOI: 10.1090/s0025-5718-99-01027-3.

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

Elliott, J. and J. Peraire (1996). “Aerodynamic design using unstructured meshes”. In: *Fluid Dynamics Conference*. 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.

Errico, R. M. (1997). “What Is an Adjoint Model?” In: *Bulletin of the American Meteorological Society* 78.11, pp. 2577–2591. DOI: 10.1175/1520-0477(1997)078<2577:wiaam>2.0.co;2.

Eyink, G., T. Haine, and D. Lea (2004). “Ruelle’s linear response formula, ensemble adjoint schemes and Lévy flights”. In: *Nonlinearity* 17.5, p. 1867.

Farrell, B. (1988). “Optimal Excitation of Neutral Rossby Waves”. In: *Journal of the Atmospheric Sciences* 45.2, pp. 163–172. DOI: 10.1175/1520-0469(1988)045<0163:oeonrw>2.0.co;2.

Farrell, B. F. and P. J. Ioannou (1996). “Generalized Stability Theory. Part I: Autonomous Operators”. In: *Journal of the Atmospheric Sciences* 53.14, pp. 2025–2040. DOI: 10.1175/1520-0469(1996)053<2025:gstpia>2.0.co;2.

Farrell, P. E., D. A. Ham, S. W. Funke, and M. E. Rognes (2013). “Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs”. In: *SIAM Journal on Scientific Computing* 35.4, pp. C369–C393. DOI: 10.1137/120873558.

Ferreira, D., J. Marshall, and P. Heimbach (2005). “Estimating Eddy Stresses by Fitting Dynamics to Observations Using a Residual-Mean Ocean Circulation Model and Its Adjoint”. In: *Journal of Physical Oceanography* 35.10, pp. 1891–1910. DOI: 10.1175/jpo2785.1.

Fike, J. A. (2013). “Multi-objective optimization using hyper-dual numbers”. PhD thesis.

Forget, G., J.-M. Campin, P. Heimbach, C. N. Hill, R. M. Ponte, and C. Wunsch (2015). “ECCO version 4: an integrated framework for non-linear inverse modeling and global ocean state estimation”. In: *Geoscientific Model Development* 8.10, pp. 3071–3104. DOI: 10.5194/gmd-8-3071-2015.

Fornberg, B. (1988). “Generation of Finite Difference Formulas on Arbitrarily Spaced Grids”. In: *Mathematics of Computation* 51.184, pp. 699–706.

Frank, S. A. (2022). “Automatic Differentiation and the Optimization of Differential Equation Models in Biology”. In: *Frontiers in Ecology and Evolution* 10.

Franklin, O., S. P. Harrison, R. Dewar, C. E. Farrior, Å. Brännström, U. Dieckmann, S. Pietsch, D. Falster, W. Cramer, M. Loreau, H. Wang, A. Mäkelä, K. T. Rebel, E. Meron, S. J. Schymanski, E. Rovenskaya, B. D. Stocker, S. Zaehle, S. Manzoni, M. van Oijen, I. J. Wright, P. Ciais, P. M. van Bodegom, J. Peñuelas, F. Hofhansl, C. Terrer, N. A. Soudzilovskaia, G. Midgley, and I. C. Prentice (2020). “Organizing Principles for Vegetation Dynamics”. In: *Nature Plants* 6.5, pp. 444–453. DOI: 10.1038/s41477-020-0655-x.

Frolov, S., C. S. Rousseaux, T. Auligne, D. Dee, R. Gelaro, P. Heimbach, I. Simpson, and L. Slivinski (2023). “Road Map for the Next Decade of Earth System Reanalysis in the United States”. In: *Bulletin of the American Meteorological Society* 104.3, E706–E714. DOI: 10.1175/bams-d-23-0011.1.

Gábor, A. and J. R. Banga (Dec. 29, 2015). “Robust and Efficient Parameter Estimation in Dynamic Models of Biological Systems”. In: *BMC Systems Biology* 9.1, p. 74. DOI: 10.1186/s12918-015-0219-2.

Gaikwad, S. S., L. Hascoet, S. H. K. Narayanan, L. Curry-Logan, R. Greve, and P. Heimbach (2023). “SICOPOLIS-AD v2: tangent linear and adjoint modeling framework for ice sheet modeling enabled by automatic differentiation tool Tapenade”. In: *Journal of Open Source Software* 8.83, p. 4679. DOI: 10.21105/joss.04679.

Gaikwad, S. S., S. H. K. Narayanan, L. Hascoet, J.-M. Campin, H. Pillar, A. Nguyen, J. Hückelheim, P. Hovland, and P. Heimbach (2024). "MITgcm-AD v2: Open source tangent linear and adjoint modeling framework for the oceans and atmosphere enabled by the Automatic Differentiation tool Tapenade". In: *arXiv*.

GBIF: The Global Biodiversity Information Facility (2022). "What Is GBIF?" In.

Geary, W. L., M. Bode, T. S. Doherty, E. A. Fulton, D. G. Nimmo, A. I. T. Tulloch, V. J. D. Tulloch, and E. G. Ritchie (Nov. 14, 2020). "A Guide to Ecosystem Models and Their Environmental Applications". In: *Nature Ecology & Evolution* 4.11, pp. 1459–1471. DOI: 10.1038/s41559-020-01298-8.

Gebremedhin, A. H., F. Manne, and A. Pothen (2005). "What color is your Jacobian? Graph coloring for computing derivatives". In: *SIAM review* 47.4, pp. 629–705.

Gelbrecht, M., A. White, S. Bathiany, and N. Boers (2023). "Differentiable programming for Earth system modeling". In: *Geoscientific Model Development* 16.11, pp. 3123–3135. DOI: 10.5194/gmd-16-3123-2023.

Gelman, A., J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin (2013). *Bayesian data analysis*. CRC press.

Georgieva, N. K., S. Glavic, M. H. Bakr, and J. W. Bandler (2002). "Feasible Adjoint Sensitivity Technique for Em Design Optimization". In: *IEEE Transactions on Microwave Theory and Techniques* 50.12, p. 2751. DOI: 10.1109/tmtt.2002.805131.

Gharamti, M., J. Tjiputra, I. Bethke, A. Samuelsen, I. Skjelvan, M. Bentsen, and L. Bertino (Apr. 2017). "Ensemble Data Assimilation for Ocean Biogeochemical State and Parameter Estimation at Different Sites". In: *Ocean Modelling* 112, pp. 65–89. DOI: 10.1016/j.ocemod.2017.02.006.

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

Giering, R. and T. Kaminski (1998a). "Recipes for adjoint code construction". In: *ACM Trans Math Softw* 24.4, pp. 437–474. DOI: 10.1145/293686.293695.

— (1998b). "Recipes for adjoint code construction". In: *ACM Transactions on Mathematical Software (TOMS)* 24.4, pp. 437–474. DOI: 10.1145/293686.293695.

Giering, R., T. Kaminski, R. Todling, R. Errico, R. Gelaro, and N. Winslow (2006). "Automatic Differentiation: Applications, Theory, and Implementations". In: *Lecture Notes in Computational Science and Engineering*, pp. 275–284. DOI: 10.1007/3-540-28438-9\\_24.

Giles, M. B. and N. A. Pierce (2000a). "An introduction to the adjoint approach to design". In: *Flow, Turbulence and Combustion* 65.3, pp. 393–415.

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

Givoli, D. (2021). "A tutorial on the adjoint method for inverse problems". In: 380, p. 113810. DOI: 10.1016/j.cma.2021.113810.

Godwin, C. M., F.-H. Chang, and B. J. Cardinale (2020). "An Empiricist's Guide to Modern Coexistence Theory for Competitive Communities". In: *Oikos* 129.8, pp. 1109–1127. DOI: 10.1111/oik.06957.

Goerz, M. H., S. C. Carrasco, and V. S. Malinovsky (2022). "Quantum optimal control via semi-automatic differentiation". In: *Quantum* 6, p. 871. DOI: 10.22331/q-2022-12-07-871.

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

Goldberg, D. and P. Heimbach (2013). "Parameter and state estimation with a time-dependent adjoint marine ice sheet model". In: *The Cryosphere* 7.6, pp. 1659–1678.

Gonnet, P. (2012). "A review of error estimation in adaptive quadrature". In: *ACM Computing Surveys (CSUR)* 44.4, pp. 1–36.

Gorban, A. and D. Wunsch (1998). "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, 1271–1274 vol.2. DOI: 10.1109/IJCNN.1998.685957.

Gowda, S., Y. Ma, A. Cheli, M. Gwóźzdź, V. B. Shah, A. Edelman, and C. Rackauckas (2022). "High-performance symbolic-numerics via multiple dispatch". In: *ACM Communications in Computer Algebra* 55.3, pp. 92–96. DOI: 10.1145/3511528.3511535.

Griewank, A. (1989). "On Automatic Differentiation". In: *Mathematical Programming: Recent Developments and Applications*.

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

Griewank, A. and A. Walther (2008). *Evaluating Derivatives*. DOI: 10.1137/1.9780898717761.

Gronwall, T. H. (1919). "Note on the derivatives with respect to a parameter of the solutions of a system of differential equations". In: *Annals of Mathematics*, pp. 292–296.

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

Hairer, E., G. Wanner, and S. Nørsett (2008). *Solving Ordinary Differential Equations I: Nonstiff Problems (Second Revised Edition)*. Springer Berlin Heidelberg New York.

Hartig, F., J. Dyke, T. Hickler, S. I. Higgins, R. B. O'Hara, S. Scheiter, and A. Huth (Dec. 2012). "Connecting Dynamic Vegetation Models to Data - an Inverse Perspective: Dynamic Vegetation Models - an Inverse Perspective". In: *Journal of Biogeography* 39.12, pp. 2240–2252. DOI: 10.1111/j.1365-2699.2012.02745.x.

Hascoet, L. and V. Pasqual (2013). "The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification". In: *ACM Transactions on Mathematical Software* 39.3, pp. 1–43. DOI: 10.1145/2450153.2450158.

Hascoët, L. and M. Morlighem (2018). "Source-to-source adjoint Algorithmic Differentiation of an ice sheet model written in C". In: *Optimization Methods and Software* 33.4-6, pp. 829–843.

Hastie, T., R. Tibshirani, J. H. Friedman, and J. H. Friedman (2009). *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer.

Heimbach, P., D. Menemenlis, M. Losch, J.-M. Campin, and C. Hill (2010). "On the formulation of sea-ice models. Part 2: Lessons from multi-year adjoint sea-ice export sensitivities through the Canadian Arctic Archipelago". In: *Ocean Modelling* 33.1-2, pp. 145–158. DOI: 10.1016/j.ocemod.2010.02.002.

Heimbach, P. and V. Bugnion (2009). "Greenland ice-sheet volume sensitivity to basal, surface and initial conditions derived from an adjoint model". In: *Annals of Glaciology* 50.52, pp. 67–80.

Heimbach, P., C. Hill, and R. Giering (2005). "An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation". In: *Future Generation Computer Systems* 21.8, pp. 1356–1371. DOI: 10.1016/j.future.2004.11.010.

Heimbach, P. and M. Losch (2012). "Adjoint sensitivities of sub-ice-shelf melt rates to ocean circulation under the Pine Island Ice Shelf, West Antarctica". In: *Annals of Glaciology* 53.60, pp. 59–69. DOI: 10.3189/2012/aog60a025.

Higgins, S. I., S. Scheiter, and M. Sankaran (June 2010). "The Stability of African Savannas: Insights from the Indirect Estimation of the Parameters of a Dynamic Model". In: *Ecology* 91.6, pp. 1682–1692. DOI: 10.1890/08-1368.1.

Hindmarsh, A. C., P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward (2005). "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3, pp. 363–396.

Hu, R. (2010). "Supersonic biplane design via adjoint method". PhD thesis.

Huang, J., T. M. Smith, G. M. Henry, and R. A. V. D. Geijn (2016). "Strassen's Algorithm Reloaded". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 690–701. DOI: 10.1109/sc.2016.58.

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

Innes, M., A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt (2019). "A Differentiable Programming System to Bridge Machine Learning and Scientific Computing". In: *arXiv*. DOI: 10.48550/arxiv.1907.07587.

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

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

Jetz, W., M. A. McGeoch, R. Guralnick, S. Ferrier, J. Beck, M. J. Costello, M. Fernandez, G. N. Geller, P. Keil, C. Merow, C. Meyer, F. E. Muller-Karger, H. M. Pereira, E. C. Regan, D. S. Schmeller, and E. Turak (2019). "Essential Biodiversity Variables for Mapping and Monitoring Species Populations". In: *Nature Ecology and Evolution* 3.4, pp. 539–551. DOI: 10.1038/s41559-019-0826-1.

Jirari, H. (2009). "Optimal control approach to dynamical suppression of decoherence of a qubit". In: *Europhysics Letters* 87.4, p. 40003. DOI: 10.1209/0295-5075/87/40003.

— (2019). "From quantum optimal control theory to coherent destruction of tunneling". In: *The European Physical Journal B* 92, pp. 1–8. DOI: 10.1140/epjb/e2018-90231-5.

Johnson, J. B. and K. S. Omland (Feb. 2004). "Model Selection in Ecology and Evolution". In: *Trends in Ecology & Evolution* 19.2, pp. 101–108. DOI: 10.1016/j.tree.2003.10.013.

Johnson, S. G. (2012). "Notes on Adjoint Methods for 18.335". In.

Jouvet, G. (2023). "Inversion of a Stokes glacier flow model emulated by deep learning". In: *Journal of Glaciology* 69.273, pp. 13–26.

Jouvet, G., G. Cordonnier, B. Kim, M. Lüthi, A. Vieli, and A. Aschwanden (2021). "Deep learning speeds up ice flow modelling by several orders of magnitude". In: *Journal of Glaciology*, pp. 1–14. DOI: 10.1017/jog.2021.120.

Juedes, D. W. (1991). *A taxonomy of automatic differentiation tools*. Tech. rep. Argonne National Lab., IL (United States).

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

Karczmarczuk, J. (1998). "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, pp. 195–203. DOI: 10.1145/289423.289442.

Kidger, P. (2021). "On Neural Differential Equations". PhD thesis. University of Oxford.

Kiehl, J. T., C. A. Shields, J. J. Hack, and W. D. Collins (2006). *The climate sensitivity of the Community Climate System Model version 3 (CCSM3)*. Journal of Climate.

Kim, S., W. Ji, S. Deng, Y. Ma, and C. Rackauckas (Sept. 2021). “Stiff neural ordinary differential equations”. en. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 31.9, p. 093122. DOI: 10.1063/5.0060697.

Kingma, D. P. and J. Ba (Dec. 22, 2014). “Adam: A Method for Stochastic Optimization”.

Kochkov, D., J. Yuval, I. Langmore, P. Norgaard, J. Smith, G. Mooers, J. Lottes, S. Rasp, P. Düben, M. Klöwer, S. Hatfield, P. Battaglia, A. Sanchez-Gonzalez, M. Willson, M. P. Brenner, and S. Hoyer (2023). “Neural General Circulation Models”. In: *arXiv*. DOI: 10.48550/arxiv.2311.07222.

Langland, R. H. and N. L. Baker (2004). “Estimation of observation impact using the NRL atmospheric variational data assimilation adjoint system”. In: *Tellus A: Dynamic Meteorology and Oceanography* 56.3, pp. 189–201. DOI: 10.3402/tellusa.v56i3.14413.

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

Laue, S. (2019). *On the Equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation*. DOI: 10.48550/ARXIV.1904.02990.

Laurie, D. (1997). “Calculation of Gauss-Kronrod quadrature rules”. In: *Mathematics of Computation* 66.219, pp. 1133–1145.

Lea, D. J., M. R. Allen, and T. W. N. Haine (2000a). “Sensitivity analysis of the climate of a chaotic system”. In: *Tellus A: Dynamic Meteorology and Oceanography* 52.5, pp. 523–532. DOI: 10.3402/tellusa.v52i5.12283.

— (2000b). “Sensitivity analysis of the climate of a chaotic system”. In: *Tellus A: Dynamic Meteorology and Oceanography* 52.5, pp. 523–532. DOI: https://doi.org/10.3402/tellusa.v52i5.12283.

Lea, D. J., T. W. N. Haine, M. R. Allen, and J. A. Hansen (2002). “Sensitivity analysis of the climate of a chaotic ocean circulation model”. In: *Quarterly Journal of the Royal Meteorological Society* 128.586, pp. 2587–2605. DOI: 10.1256/qj.01.180.

LeCun, Y., Y. Bengio, and G. Hinton (May 27, 2015). “Deep Learning”. In: *Nature* 521.7553, pp. 436–444. DOI: 10.1038/nature14539.

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

Leith, C. E. (1975). “Climate response and fluctuation dissipation”. In: *Journal of Atmospheric Sciences* 32.10, pp. 2022–2026. DOI: https://doi.org/10.1175/1520-0469(1975)032<2022:CRAFD>2.0.CO;2.

Leung, N., M. Abdelhafez, J. Koch, and D. Schuster (2017). “Speedup for quantum optimal control from automatic differentiation based on graphics processing units”. In: *Physical Review A* 95.4, p. 042318. DOI: 10.1103/PhysRevA.95.042318.

Li, D., K. Xu, J. M. Harris, and E. Darve (2020a). “Coupled time-lapse full-waveform inversion for subsurface flow problems using intrusive automatic differentiation”. In: *Water Resources Research* 56.8, e2019WR027032.

Li, X., T.-K. L. Wong, R. T. Chen, and D. Duvenaud (2020b). “Scalable gradients for stochastic differential equations”. In: *International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 3870–3882.

Lion, S. (2018). “Theoretical Approaches in Evolutionary Ecology: Environmental Feedback as a Unifying Perspective”. In: *American Naturalist* 191.1, pp. 21–44. DOI: 10.1086/694865.

Lions, J. L. (1971). *Optimal control of systems governed by partial differential equations*. Vol. 170. Springer.

Liu, C., A. Köhl, and D. Stammer (2012). "Adjoint-Based Estimation of Eddy-Induced Tracer Mixing Parameters in the Global Ocean". In: *Journal of Physical Oceanography* 42.7, pp. 1186–1206. DOI: 10.1175/jpo-d-11-0162.1.

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

Lyness, J. N. and C. B. Moler (1967). "Numerical Differentiation of Analytic Functions". In: *SIAM Journal on Numerical Analysis* 4.2, pp. 202–210. DOI: 10.1137/0704019.

Lyu, G., A. Köhl, I. Matei, and D. Stammer (2018). "Adjoint-Based Climate Model Tuning: Application to the Planet Simulator". In: *Journal of Advances in Modeling Earth Systems* 10.1, pp. 207–222. DOI: 10.1002/2017ms001194.

Ma, Y., V. Dixit, M. J. Innes, X. Guo, and C. Rackauckas (2021). "A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions". In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, pp. 1–9.

MacAyeal, D. R. (1992). "The basal stress distribution of Ice Stream E, Antarctica, inferred by control methods". In: *Journal of Geophysical Research: Solid Earth* 97.B1, pp. 595–603.

Manzyuk, O., B. A. Pearlmutter, A. A. Radul, D. R. Rush, and J. M. Siskind (2019). "Perturbation confusion in forward automatic differentiation of higher-order functions". In: *Journal of Functional Programming* 29, e12.

Margossian, C. C. (2018). "A Review of automatic differentiation and its efficient implementation". In: *arXiv*. DOI: 10.48550/arxiv.1811.05031.

Marotzke, J., R. Giering, K. Q. Zhang, D. Stammer, C. Hill, and T. Lee (1999). "Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport sensitivity". In: *Journal of Geophysical Research* 104.29, pp. 529–548. DOI: 10.1029/1999jc900236.

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

Martins, J., P. Sturdza, and J. Alonso (2001). "The connection between the complex-step derivative approximation and algorithmic differentiation". In: *39th Aerospace Sciences Meeting and Exhibit*, p. 921.

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

McGreivy, N., S. Hudson, and C. Zhu (2021). "Optimized finite-build stellarator coils using automatic differentiation". In: *Nuclear Fusion* 61, p. 026020. DOI: 10.1088/1741-4326/abcd76.

Meehl, G. A., J. H. Richter, H. Teng, A. Capotondi, K. Cobb, F. Doblas-Reyes, M. G. Donat, M. H. England, J. C. Fyfe, W. Han, H. Kim, B. P. Kirtman, Y. Kushnir, N. S. Lovenduski, M. E. Mann, W. J. Merryfield, V. Nieves, K. Pegion, N. Rosenbloom, S. C. Sanchez, A. A. Scaife, D. Smith, A. C. Subramanian, L. Sun, D. Thompson, C. C. Ummenhofer, and S.-P. Xie (2021). "Initialized Earth System prediction from subseasonal to decadal timescales". In: *Nature Reviews Earth & Environment*, pp. 1–18. DOI: 10.1038/s43017-021-00155-x.

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

Mohammadi, B. and O. Pironneau (2004). “Shape optimization in fluid mechanics”. In: *Annual Review of Fluid Mechanics* 36.1, pp. 255–279. DOI: 10.1146/annurev.fluid.36.050802.121926.

— (2009). *Applied shape optimization for fluids*. OUP Oxford.

Molesky, S., Z. Lin, A. Y. Piggott, W. Jin, J. Vucković, and A. W. Rodriguez (2018). “Inverse design in nanophotonics”. In: *Nature Photonics* 12.11, pp. 659–670. DOI: 10.1038/s41566-018-0246-9.

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

Moore, A. M. and R. Kleeman (1997a). “The singular vectors of a coupled ocean-atmosphere model of Enso. I: Thermodynamics, energetics and error growth”. In: *Quarterly Journal of the Royal Meteorological Society* 123.540, pp. 953–981. DOI: 10.1002/qj.49712354009.

— (1997b). “The singular vectors of a coupled ocean-atmosphere model of Enso. II: Sensitivity studies and dynamical interpretation”. In: *Quarterly Journal of the Royal Meteorological Society* 123.540, pp. 983–1006. DOI: 10.1002/qj.49712354010.

Morlighem, M., H. Seroussi, E. Larour, and E. Rignot (2013). “Inversion of basal friction in Antarctica using exact and incomplete adjoints of a higher-order model”. In: *Journal of Geophysical Research: Earth Surface* 118.3, pp. 1746–1753.

Mosbeux, C., F. Gillet-Chaulet, and O. Gagliardini (2016). “Comparison of adjoint and nudging methods to initialise ice sheet model basal conditions”. In: *Geoscientific Model Development* 9.7, pp. 2549–2562.

Moses, W. and V. Churavy (2020). “Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., pp. 12472–12485.

Moses, W. S., V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert (2021). “Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme”. In: *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis* 00, pp. 1–18. DOI: 10.1145/3458817.3476165.

Muehlebach, M. and M. I. Jordan (2021). “Optimization with Momentum: Dynamical, Control-Theoretic, and Symplectic Perspectives”. In: *Journal of Machine Learning Research* 22.73, pp. 1–50.

Murphy, K. P. (2022). *Probabilistic Machine Learning: An introduction*. MIT Press.

Naumann, U. (2011). *The Art of Differentiating Computer Programs*. SIAM. DOI: 10.1137/1.9781611972078.

Neal, R. M. et al. (2011). “MCMC using Hamiltonian dynamics”. In: *Handbook of markov chain monte carlo* 2.11, p. 2.

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

Ni, A. (2020). “Fast linear response algorithm for differentiating chaos”. In: *arXiv preprint arXiv:2009.00595*. DOI: https://arxiv.org/abs/2009.00595v5.

Ni, A. and C. Talnikar (2019a). “Adjoint sensitivity analysis on chaotic dynamical systems by Non-Intrusive Least Squares Adjoint Shadowing (NILSAS)”. In: *Journal of Computational Physics* 395, pp. 690–709. DOI: https://doi.org/10.1016/j.jcp.2019.06.035.

Ni, A. and Q. Wang (2017). "Sensitivity analysis on chaotic dynamical systems by Non-Intrusive Least Squares Shadowing (NILSS)". In: *Journal of Computational Physics* 347, pp. 56–77. DOI: https://doi.org/10.1016/j.jcp.2017.06.033.

Ni, A., Q. Wang, P. Fernandez, and C. Talnikar (2019b). "Sensitivity analysis on chaotic dynamical systems by finite difference non-intrusive least squares shadowing (FD-NILSS)". In: *Journal of Computational Physics* 394, pp. 615–631. DOI: https://doi.org/10.1016/j.jcp.2019.06.004.

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

Nurbekyan, L., W. Lei, and Y. Yang (2023). "Efficient Natural Gradient Descent Methods for Large-Scale PDE-Based Optimization Problems". In: *SIAM Journal on Scientific Computing* 45.4, A1621–A1655. DOI: 10.1137/22M1477805.

Oktay, D., N. McGreivy, J. Aduol, A. Beatson, and R. P. Adams (2020). "Randomized Automatic Differentiation". In: *arXiv*. DOI: 10.48550/arxiv.2007.10412.

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

PAGE, K. M. and M. A. NOWAK (Nov. 2002). "Unifying Evolutionary Dynamics". In: *Journal of Theoretical Biology* 219.1, pp. 93–98. DOI: 10.1006/jtbi.2002.3112.

Palmer, T. N., R. Buizza, F. Molteni, Y.-Q. Chen, and S. Corti (1994). "Singular vectors and the predictability of weather and climate". In: *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences* 348.1688, pp. 459–475. DOI: 10.1098/rsta.1994.0105.

Palmieri, G., M. Tiboni, and G. Legnani (2020). "Analysis of the Upper Limitation of the Most Convenient Cadence Range in Cycling Using an Equivalent Moment Based Cost Function". In: *Mathematics* 8.11. DOI: 10.3390/math8111947.

Pantel, J. H. and L. Becks (June 2023). "Statistical Methods to Identify Mechanisms in Studies of Eco-Evolutionary Dynamics". In: *Trends in Ecology & Evolution*, S0169534723000800. DOI: 10.1016/j.tree.2023.03.011.

Paredes, J. A., K. Hufkens, and B. D. Stocker (Nov. 24, 2023). *Rsofun: A Model-Data Integration Framework for Simulating Ecosystem Processes*. DOI: 10.1101/2023.11.24.568574. URL: https://www.biorxiv.org/content/10.1101/2023.11.24.568574v1 (visited on 11/28/2023). preprint.

Pironneau, O. (2005). "Optimal shape design for elliptic systems". In: *System Modeling and Optimization: Proceedings of the 10th IFIP Conference New York City, USA, August 31–September 4, 1981*. Springer, pp. 42–66.

Pontarp, M., Å. Brännström, and O. L. Petchey (Apr. 2019). "Inferring Community Assembly Processes from Macroscopic Patterns Using Dynamic Eco-evolutionary Models and Approximate Bayesian Computation (ABC)". In: *Methods in Ecology and Evolution* 10.4. Ed. by T. Poisot, pp. 450–460. DOI: 10.1111/2041-210X.13129.

Rabier, F., H. Järvinen, E. Klinker, J. F. Mahfouf, and A. Simmons (2000). "The ECMWF operational implementation of four-dimensional variational assimilation. I: Experimental results with simplified physics". In: *Quarterly Journal of the Royal Meteorological Society* 126.564, pp. 1143–1170. DOI: 10.1002/qj.49712656415.

Rabier, F. and P. Courtier (1992). "Four-Dimensional Assimilation In the Presence of Baroclinic Instability". In: *Quarterly Journal of the Royal Meteorological Society* 118.506, pp. 649–672. DOI: 10.1002/qj.49711850604.

Rackauckas, C., A. Edelman, K. Fischer, M. Innes, E. Saba, V. B. Shah, and W. Tebbutt (2021). “Generalized physics-informed learning through language-wide differentiable programming”. In.

Rackauckas, C., Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman (2020). “Universal differential equations for scientific machine learning”. In: *arXiv preprint arXiv:2001.04385*.

Rackauckas, C. and Q. Nie (2016). “DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia”. In: *Journal of Open Research Software* 5.1, p. 15. DOI: 10.5334/jors.151.

Raissi, M., P. Perdikaris, and G. Karniadakis (2019). “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, pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045.

Ramsay, J. and G. Hooker (2017). *Dynamic Data Analysis, Modeling Data with Differential Equations*. DOI: 10.1007/978-1-4939-7190-9.

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

Ranocha, H., L. Dalcin, M. Parsani, and D. I. Ketcheson (2022). “Optimized Runge-Kutta Methods with Automatic Step Size Control for Compressible Computational Fluid Dynamics”. In: *Communications on Applied Mathematics and Computation* 4.4. Paper with the RDPK3Sp35 method, pp. 1191–1228. DOI: 10.1007/s42967-021-00159-w.

Rasp, S., M. S. Pritchard, and P. Gentine (Sept. 25, 2018). “Deep Learning to Represent Subgrid Processes in Climate Models”. In: *Proceedings of the National Academy of Sciences* 115.39, pp. 9684–9689. DOI: 10.1073/pnas.1810286115.

Razavi, S., A. Jakeman, A. Saltelli, C. Prieur, B. Iooss, E. Borgonovo, E. Plischke, S. L. Piano, T. Iwanaga, W. Becker, S. Tarantola, J. H. A. Guillaume, J. Jakeman, H. Gupta, N. Melillo, G. Rabitti, V. Chabridon, Q. Duan, X. Sun, S. Smith, R. Sheikholeslami, N. Hosseini, M. Asadzadeh, A. Puy, S. Kucherenko, and H. R. Maier (2021). “The Future of Sensitivity Analysis: An essential discipline for systems modeling and policy support”. In: *Environmental Modelling & Software* 137, p. 104954. DOI: 10.1016/j.envsoft.2020.104954.

Revels, J., M. Lubin, and T. Papamarkou (2016). “Forward-Mode Automatic Differentiation in Julia”. In: *arXiv:1607.07892 [cs.MS]*.

Rosenbaum, B., M. Raatz, G. Weithoff, G. F. Fussmann, and U. Gaedke (Jan. 22, 2019). “Estimating Parameters From Multiple Time Series of Population Dynamics Using Bayesian Inference”. In: *Frontiers in Ecology and Evolution* 6 (JAN). DOI: 10.3389/fevo.2018.00234.

Rüde, U., K. Willcox, L. C. McInnes, and H. D. Sterck (2018). “Research and Education in Computational Science and Engineering”. In: *SIAM Review* 60.3, pp. 707–754. DOI: 10.1137/16m1096840.

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

Rudin, C., C. Chen, Z. Chen, H. Huang, L. Semenova, and C. Zhong (2022). “Interpretable machine learning: Fundamental principles and 10 grand challenges”. In: *Statistic Surveys* 16.none, pp. 1–85. DOI: 10.1214/21-ss133.

Ruelle, D. (1997). “Differentiation of SRB states”. In: *Communications in Mathematical Physics* 187.1, pp. 227–241. DOI: https://doi.org/10.1007/s002200050134.

— (2009). “A review of linear response theory for general differentiable dynamical systems”. In: *Nonlinearity* 22.4, p. 855. DOI: https://iopscience.iop.org/article/10.1088/0951-7715/22/4/009.

Ruppert, K. M., R. J. Kline, and M. S. Rahman (Jan. 2019). “Past, Present, and Future Perspectives of Environmental DNA (eDNA) Metabarcoding: A Systematic Review in Methods, Monitoring, and Applications of Global eDNA”. In: *Global Ecology and Conservation* 17, e00547. DOI: 10.1016/j.gecco.2019.e00547.

Sandu, A. (2006). “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, pp. 550–557.

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

Schäfer, F., M. Kloc, C. Bruder, and N. Lörch (2020). “A differentiable programming method for quantum control”. In: *Machine Learning: Science and Technology* 1.3, p. 035009. DOI: 10.1088/2632-2153/ab9802.

Schäfer, F., P. Sekatski, M. Koppenhöfer, C. Bruder, and M. Kloc (2021a). “Control of stochastic quantum dynamics by differentiable programming”. In: *Machine Learning: Science and Technology* 2.3, p. 035004. DOI: 10.1088/2632-2153/abec22.

Schäfer, F., M. Tarek, L. White, and C. Rackauckas (2021b). “AbstractDifferentiation.jl: Backend-Agnostic Differentiable Programming in Julia”. In: *arXiv*. DOI: 10.48550/arxiv.2109.12449.

Schanen, M., S. H. K. Narayanan, S. Williamson, V. Churavy, W. S. Moses, and L. Paehler (2023). “Transparent Checkpointing for Automatic Differentiation of Program Loops Through Expression Transformations”. In: ed. by J. Mikyška, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. Sloot, pp. 483–497.

Schartau, M., P. Wallhead, J. Hemmings, U. Löptien, I. Kriest, S. Krishna, B. A. Ward, T. Slawig, and A. Oschlies (Mar. 29, 2017). “Reviews and Syntheses: Parameter Identification in Marine Planktonic Ecosystem Modelling”. In: *Biogeosciences* 14.6, pp. 1647–1701. DOI: 10.5194/bg-14-1647-2017.

Schneider, T., S. Lan, A. Stuart, and J. Teixeira (Dec. 28, 2017). “Earth System Modeling 2.0: A Blueprint for Models That Learn From Observations and Targeted High-Resolution Simulations”. In: *Geophysical Research Letters* 44.24, pp. 12, 396–12, 417. DOI: 10.1002/2017GL076101.

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

Shen, C., A. P. Appling, P. Gentine, T. Bandai, H. Gupta, A. Tartakovsky, M. Baity-Jesi, F. Fenicia, D. Kifer, L. Li, X. Liu, W. Ren, Y. Zheng, C. J. Harman, M. Clark, M. Farthing, D. Feng, P. Kumar, D. Aboelyazeed, F. Rahmani, Y. Song, H. E. Beck, T. Bindas, D. Dwivedi, K. Fang, M. Höge, C. Rackauckas, B. Mohanty, T. Roy, C. Xu, and K. Lawson (2023). “Differentiable modelling to unify machine learning and physical models for geosciences”. In: *Nature Reviews Earth & Environment*, pp. 1–16. DOI: 10.1038/s43017-023-00450-9.

Silva, H. D., J. L. Gustafson, and W.-F. Wong (2018). “Making Strassen Matrix Multiplication Safe”. In: *2018 IEEE 25th International Conference on High Performance Computing (HiPC)* 00, pp. 173–182. DOI: 10.1109/hipc.2018.00028.

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

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

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

Stammer, D., A. Köhl, A. Vlasenko, I. Matei, F. Lunkeit, and S. Schubert (2018). "A Pilot Climate Sensitivity Study Using the CEN Coupled Adjoint Model (CESAM)". In: *Journal of Climate* 31.5, pp. 2031–2056. DOI: 10.1175/jcli-d-17-0183.1.

Stammer, D., C. Wunsch, R. Giering, C. Eckert, P. Heimbach, J. Marotzke, A. Adcroft, C. N. Hill, and J. Marshall (2002). "Global ocean circulation during 1992–1997, estimated from ocean observations and a general circulation model". In: *Journal of Geophysical Research: Oceans* 107.C9, pp. 1–1-1-27. DOI: 10.1029/2001jc000888.

Stammer, D. (2005). "Adjusting Internal Model Errors through Ocean State Estimation". In: *Journal of Physical Oceanography* 35.6, pp. 1143–1153. DOI: 10.1175/jpo2733.1.

Stein, E. M. and R. Shakarchi (2010). *Complex analysis.* Vol. 2. Princeton University Press.

Strouwen, A., B. M. Nicolaï, and P. Goos (Jan. 12, 2022). "Robust Dynamic Experiments for the Precise Estimation of Respiration and Fermentation Parameters of Fruit and Vegetables". In: *PLOS Computational Biology* 18.1. Ed. by P. Mendes, e1009610. DOI: 10.1371/journal.pcbi.1009610.

Talagrand, O. and P. Courtier (1987). "Variational Assimilation of Meteorological Observations With the Adjoint Vorticity Equation. I: Theory". In: *Quarterly Journal of the Royal Meteorological Society* 113.478, pp. 1311–1328. DOI: 10.1002/qj.49711347812.

Thacker, W. C. (1988). "Fitting models to inadequate data by enforcing spatial and temporal smoothness". In: *Journal of Geophysical Research: Oceans (1978–2012)* 93.C9, pp. 10655–10665. DOI: 10.1029/jc093ic09p10655.

— (1989). "The role of the Hessian matrix in fitting models to measurements". In: *Journal of Geophysical Research: Oceans (1978–2012)* 94.C5, pp. 6177–6196. DOI: 10.1029/jc094ic05p06177.

Thacker, W. C. and R. B. Long (1988). "Fitting dynamics to data". In: *Journal of Geophysical Research: Oceans (1978–2012)* 93.C2, pp. 1227–1240. DOI: 10.1029/jc093ic02p01227.

Thuburn, J. (2005). "Climate sensitivities via a Fokker–Planck adjoint approach". In: *Quarterly Journal of the Royal Meteorological Society: A journal of the atmospheric sciences, applied meteorology and physical oceanography* 131.605, pp. 73–92. DOI: https://doi.org/10.1256/qj.04.46.

Toms, B. A., E. A. Barnes, and I. Ebert-Uphoff (2020). "Physically Interpretable Neural Networks for the Geosciences: Applications to Earth System Variability". In: *Journal of Advances in Modeling Earth Systems* 12.9, pp. 1–20. DOI: 10.1029/2019MS002002.

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

Tziperman, E., W. C. Thacker, and R. B. Long (1992a). "Oceanic data analysis using a general circulation model. Part I: Simulations". In: *Journal of Physical Oceanography* 22.12, pp. 1434–1457. DOI: 10.1175/1520-0485(1992)022<1434:odauag>2.0.co;2.

— (1992b). "Oceanic data analysis using a general circulation model. Part II: A North Atlantic model". In: *Journal of Physical Oceanography* 22.12, pp. 1458–1485. DOI: 10.1175/1520-0485(1992)022<1458:odauag>2.0.co;2.

Tziperman, E. and W. C. Thacker (1989). "An Optimal-Control/Adjoint-Equations Approach to Studying the Oceanic General Circulation". In: *Journal of Physical Oceanography* 19.10, pp. 1471–1485. DOI: 10.1175/1520-0485(1989)019<1471:aoceat>2.0.co;2.

Utke, J., U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch (2008). “OpenAD/F: A Modular Open-Source Tool for Automatic Differentiation of Fortran Codes”. In: *ACM Transactions on Mathematical Software (TOMS)* 34.4, p. 18. DOI: 10.1145/1377596.1377598.

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

Van den Berg, N. I., D. Machado, S. Santos, I. Rocha, J. Chacón, W. Harcombe, S. Mitri, and K. R. Patil (May 16, 2022). “Ecological Modelling Approaches for Predicting Emergent Properties in Microbial Communities”. In: *Nature Ecology & Evolution*. DOI: 10.1038/s41559-022-01746-7.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc.

Villa, C., M. A. J. Chaplain, and T. Lorenzi (Mar. 6, 2021). “Evolutionary Dynamics in Vascularised Tumours under Chemotherapy: Mathematical Modelling, Asymptotic Analysis and Numerical Simulations”. In: *Vietnam Journal of Mathematics* 49.1, pp. 143–167. DOI: 10.1007/s10013-020-00445-9.

Walther, A. (2007). “Automatic differentiation of explicit Runge-Kutta methods for optimal control”. In: *Computational Optimization and Applications* 36.1, pp. 83–108. DOI: 10.1007/s10589-006-0397-3.

Wang, F., D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf (2019). “Backpropagation with Continuation Callbacks:Foundations for Efficient and ExpressiveDifferentiable Programming”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP, p. 96. DOI: 10.1145/3341700.

Wang, Q. (2013). “Forward and adjoint sensitivity computation of chaotic dynamical systems”. In: *Journal of Computational Physics* 235, pp. 1–13. DOI: https://doi.org/10.1016/j.jcp.2012.09.007.

— (2014). “Convergence of the least squares shadowing method for computing derivative of ergodic averages”. In: *SIAM Journal on Numerical Analysis* 52.1, pp. 156–170. DOI: https://doi.org/10.1137/130917065.

Wang, Q., R. Hu, and P. Blonigan (2014a). “Least Squares Shadowing sensitivity analysis of chaotic limit cycle oscillations”. In: *Journal of Computational Physics* 267, pp. 210–224. DOI: https://doi.org/10.1016/j.jcp.2014.03.002.

— (2014b). “Least squares shadowing sensitivity analysis of chaotic limit cycle oscillations”. In: *Journal of Computational Physics* 267, pp. 210–224. DOI: https://doi.org/10.1016/j.jcp.2014.03.002.

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

Wanner, G. and E. Hairer (1996). *Solving ordinary differential equations II*. Vol. 375. Springer Berlin Heidelberg New York.

Watts, M. C. (Oct. 1, 2001). “Modelling and the Monitoring of Mesocosm Experiments: Two Case Studies”. In: *Journal of Plankton Research* 23.10, pp. 1081–1093. DOI: 10.1093/plankt/23.10.1081.

Weng, E. S., S. Malyshev, J. W. Lichstein, C. E. Farrior, R. Dybzinski, T. Zhang, E. Shevliakova, and S. W. Pacala (May 7, 2015). “Scaling from Individual Trees to Forests in an Earth System

Modeling Framework Using a Mathematically Tractable Model of Height-Structured Competition". In: *Biogeosciences* 12.9, pp. 2655–2694. DOI: 10.5194/bg-12-2655-2015.

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

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

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

Wunsch, C. (2008). "The circulation of the ocean and its variability". In: *Progress in Physical Geography* 32.4, pp. 463–474. DOI: 10.1177/0309133308096753.

Wunsch, C. and P. Heimbach (2007). "Practical global oceanic state estimation". In: *Physica D: Nonlinear Phenomena* 230.1-2, pp. 197–208. DOI: 10.1016/j.physd.2006.09.040.

Xu, P., F. Roosta, and M. W. Mahoney (n.d.). "Second-order Optimization for Non-convex Machine Learning: an Empirical Study". In: *Proceedings of the 2020 SIAM International Conference on Data Mining (SDM)*, pp. 199–207. DOI: 10.1137/1.9781611976236.23.

Yazdani, A., L. Lu, M. Raissi, and G. E. Karniadakis (Nov. 18, 2020). "Systems Biology Informed Deep Learning for Inferring Parameters and Hidden Dynamics". In: *PLOS Computational Biology* 16.11. Ed. by V. Hatzimanikatis, e1007575. DOI: 10.1371/journal.pcbi.1007575.

Zanna, L., P. Heimbach, A. M. Moore, and E. Tziperman (2012). "Upper-ocean singular vectors of the North Atlantic climate with implications for linear predictability and variability". In: *Quarterly Journal of the Royal Meteorological Society* 138.663, pp. 500–513. DOI: 10.1002/qj.937.

Zanna, L., P. Heimbach, A. M. Moore, and E. Tziperman (2011). "Optimal Excitation of Interannual Atlantic Meridional Overturning Circulation Variability". In: *Journal of Climate* 24.2, pp. 413–427. DOI: 10.1175/2010jcli3610.1.

— (2010). "The Role of Ocean Dynamics in the Optimal Growth of Tropical SST Anomalies". In: *Journal of Physical Oceanography* 40.5, pp. 983–1003. DOI: 10.1175/2009jpo4196.1.

Zdeborová, L. (May 2020). "Understanding deep learning is also a job for physicists". en. In: *Nature Physics*. DOI: 10.1038/s41567-020-0929-2.

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

Zhu, W., K. Xu, E. Darve, and G. C. Beroza (2021). "A general approach to seismic inversion with automatic differentiation". In: *Computers & Geosciences* 151, p. 104751. DOI: 10.1016/j.cageo.2021.104751.

Zhuang, J., N. Dvornek, X. Li, S. Tatikonda, X. Papademetris, and J. Duncan (2020). "Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE." In: *Proceedings of machine learning research* 119, pp. 11639–11649.

Zimmermann, N. E., T. C. Edwards Jr, C. H. Graham, P. B. Pearman, and J.-C. Svenning (2010). "New Trends in Species Distribution Modelling". In: *Ecography* 33.6, pp. 985–989. DOI: 10.1111/j.1600-0587.2010.06953.x.