A Review of Sensitivity Methods for Differential Equations

Facundo Sapienza*[1], Jordi Bolibar[2], and Fernando Pérez[1]

[1] *Department of Statistics, University of California, Berkeley (USA)*
[2] *TU Delft, Department of Geosciences and Civil Engineering, Delft (Netherlands)*

October 13, 2023

*Corresponding author: fsapienza@berkeley.edu

To the community, by the community. *This manuscript was conceived with the goal of shortening the gap between developers and practitioners of differential 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* `https://github.com/ODINN-SciML/DiffEqSensitivity-Review`

Contents

Plain language summary

Plain language summary

1 Introduction

Evaluating how the value of a function changes with respect to its arguments and parameters plays a central role in optimization, sensitivity analysis, Bayesian inference, uncertainty quantification, among many. Modern machine learning applications require the use of gradients to explore and exploit more efficiently the space of parameters. When optimizing a loss function, gradient-based methods (for example, gradient descent and its many variants [1]) are more efficient at finding a minimum and converge faster to them than than gradient-free methods. When numerically computing the posterior of a probabilistic model, gradient-based sampling strategies converge faster to the posterior distribution than gradient-free methods. Second derivatives further help to improve convergence rates of these algorithms and enable uncertainty quantification around parameter values. *A gradient serves as a compass in modern data science: it tells us in which direction in the open wide ocean of parameters we should move towards in order to increase our chances of success.*

Dynamical systems governed by differential equations are not an exception to the rule. Differential equations play a central role in describing the behaviour of systems in natural and social sciences. In order to bridge the gap between modern machine learning methods and traditional scientific methods, some authors have recently suggested differentiable programming as the solution to this problem [2, 3]. Being able to compute gradients and sensitivities of dynamical systems opens the door to more complex models. This is very appealing in geophysical models, where there is a broad literature on physical models and a long tradition in numerical methods. The first goal of this work is to introduce some of the applications of this emerging technology and to motivate its incorporation for the modelling of complex systems in the natural and social sciences.

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

There are different approaches to calculate the gradient of dynamical systems depending on the traditions of each field. 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 [4]. 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 [5], 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 Physics-Informed Neural Networks (PINNs) [6] and Universal Differential Equations (UDEs) [7].

However, when working with differential equations, the computation of gradients is not an easy task, both regarding the mathematical framework and software implementation involved. Except for a small set of particular cases, most differential equations require numerical methods to calculate their solution and cannot be differentiated analytically. This means

that solutions cannot be directly differentiated and require special treatment if, besides the numerical solution, we also want to compute first or second order derivatives. Furthermore, numerical solutions introduce approximation errors. These errors can be propagated and amplified during the computation of the gradient. Alternatively, there is a broad literature in numerical methods for solving differential equations. Although each method provides different guarantees and advantages depending on the use case, this means that the tools developed to compute gradients when using a solver need to be universal enough in order to be applied to all or at least to a large set of them. The second goal of this article is making a review of the different methods that exists to achieve this goal.

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

The broader set of tools known as Automatic Differentiation (AD) aims at computing derivatives by sequentially applying the chain rule to the sequence of unit operations that constitute a computer program. The premise is simple: every computer program, including a numerical solver, is ultimately an algorithm described by a chain of simple algebraic operations (addition, multiplication) that are easy to differentiate and their combination is easy to differentiate by using the chain rule. Although many modern differentiation tools use AD to some extend, there is also a family of methods that compute the gradient by relying on a auxiliary set of differential equations. We are going to refer to this family of methods as *continuous*, and we will dedicate them a special treatment in future sections to distinguish them from the discrete algorithms that resemble more to pure AD.

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

Despite the fact that these methods can be (in principle) implemented in different programming languages, here we decided to use the Julia programming language for the different examples. Julia is a recently new but mature programming language that has already a large tradition in implementing packages aiming to advance differential programming [8].

Without aiming at making an extensive and specialized review of the field, we believe this study will be useful to other researchers working on problems that combine optimization and sensitivity analysis with differential equations. Differential programming is opening new ways of doing research across sciences. As we make progress in the use of these tools, new methodological questions start to emerge. How do these methods compare? How can their been improved? We also hope this paper serves as a gateway to new questions regarding advances in these methods.

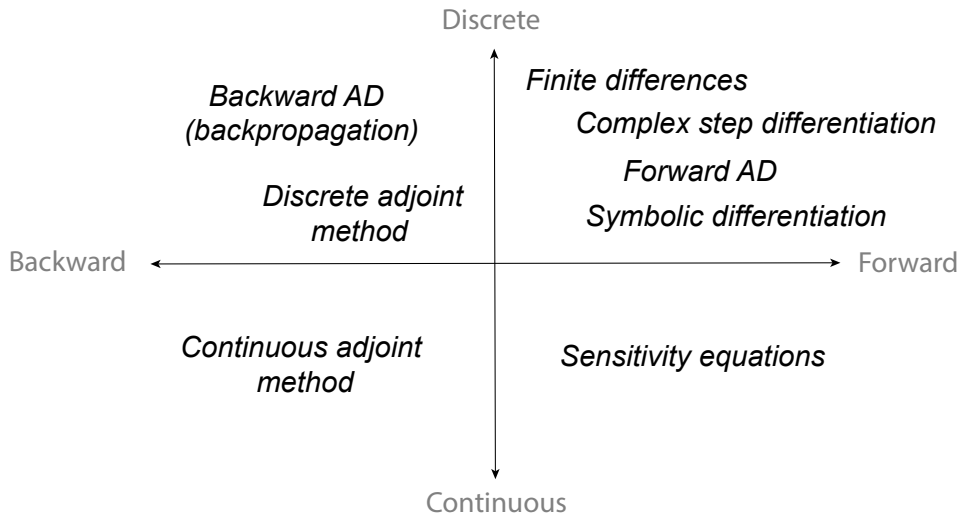

Figure 1: Schematic representation of the different methods available for differentiation involving differential equation solutions. These can be classified depending if they find the gradient by solving a new system of differential equations (*continuous*) or if instead they manipulate unit algebraic operations (*discrete*). Furthermore, depending if these methods run in the same direction than the numerical solver, we are going to be talking about *backward* and *forward* methods.

2 Scientific motivation

3 Methods

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

- *Discrete* vs *continuous* methods
- *Forward* vs *backwards* methods

The first difference regards the fact that the method for computing the gradient can be either based on the manipulation of atomic operations that are easy to differentiate using the chain rule several times (discrete), in opposition to the approach of approximating the gradient as the numerical solution of a new set of differential equations (continuous). Another way of conceptualizing this difference is by comparing them with the discretize-optimize and optimize-discretize approaches [9, 10]. We can either discretize the original system of ODEs in order to numerically solve it and then define the set of adjoint equations on top of the numerical scheme; or instead define the adjoint equation directly using the differential equation and then discretize both in order to solve [5].

The second distinction is related to the fact that some methods compute gradients by resolving a new sequential problem that may move in the same direction of the original numerical solver - i.e. moving forward in time - or, instead, they solve a new system that

goes backwards in time. Figure 1 displays a classification of some methods under this two-fold classification. In the following section we are going to explore more in detail these methods.

3.1 Preliminaries

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

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

where $u \in \mathbb{R}^n$ is the unknown solution; f is a function that depends on the state u, some parameter $\theta \in \mathbb{R}^p$, and potentially time t; and with initial condition $u(t_0) = u_0$. Here n denotes the total number of ODEs and p the size of a parameter embedded in the functional form of the differential equation. Although here we consider the case of ODEs, that is, when the derivatives are just with respect to the time variable t, this also includes the case of partial differential equations (PDEs). Furthermore, the fact that both u and θ are one-dimensional vectors does not prevent the use of higher-dimension objects (e.g. when u is a matrix or a tensor).

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

- **Loss functions**. This is usually a real valued function that quantifies the accuracy or prediction power of a given model. Examples of loss functions include the squared error

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

  where $u^{\text{target}(t_1)}$ is the desired target observation at some later time t_1; and

$$L(u(\cdot, \theta)) = \int_{t_0}^{t_1} h(u(t; \theta), \theta))dt, \tag{3}$$

  with h being a function that quantifies the contribution of the error term at every time $t \in [t_0, t_1]$.

- **Likelihood profiles.**

- **Summary of the solution.** Another important example is when L returns the value of the solution at one or many points, which is useful when we want to know how the solution itself changes as we move the parameter values.

- **Diagnosis of the solution.** In many cases we are interested in optimizing the value of some interest quantity that is a function of the solution of a differential equation. This is the case in design control theory, a popular approach in aerodynamics modelling where goals include to maximize the speed of an airplane given the solution of the flow equation for a given geometry profile [11].

We are interested in computing the gradient of the loss function with respect to the parameter θ, which can be written using the chain rule as

$$\frac{dL}{d\theta} = \frac{dL}{du}\frac{\partial u}{\partial \theta}. \tag{4}$$

The first term on the right hand side is usually easy to evaluate, since it just involves the partial derivative of the scalar loss function with respect to the solution. For example, for the loss function in Equation (2) this is simply

$$\frac{dL}{du} = u - u^{\text{target}(t_1)}. \tag{5}$$

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

$$s = \frac{\partial u}{\partial \theta} = \begin{bmatrix} \frac{\partial u_1}{\partial \theta_1} & \cdots & \frac{\partial u_1}{\partial \theta_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_n}{\partial \theta_1} & \cdots & \frac{\partial u_n}{\partial \theta_p} \end{bmatrix} \in \mathbb{R}^{n \times p}. \tag{6}$$

Notice here the distinction between the total derivative (indicated with the d) and partial derivative symbols (∂). When a function depends on more than one argument, we are going to use the partial derivative symbol to emphasize this distinction (e.g., Equation (6)). On the other side, when this is not the case, we will use the total derivative symbol (e.g., Equation (5)). Also notice that the sensitivity s defined in Equation (6) is what is called a *Jacobian*, that is, a matrix of first derivatives for general vector-valued functions.

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 a loss function, we can approximate

$$\frac{dL}{d\theta_i}(\theta) \approx \frac{L(\theta + \varepsilon e_i) - L(\theta)}{\varepsilon}, \tag{7}$$

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

$$\frac{dL}{d\theta_i}(\theta) \approx \frac{L(\theta + \varepsilon e_i) - L(\theta - \varepsilon e_i)}{2\varepsilon}, \tag{8}$$

leads also to more precise estimation of the derivative. While Equation (7) gives to an error of magnitude $\mathcal{O}(\varepsilon)$, the centered differences schemes improves to $\mathcal{O}(\varepsilon^2)$ [12].

However, there are a series of problems associated to this approach. The first one is due to how this scales with the number of parameters p. Each directional derivative requires the evaluation of the loss function L twice. For the centered differences approach in Equation (8), this requires a total of $2p$ function evaluations, which at the same time demands to solve the differential equation in forward mode each time for a new set of parameters.

A second problem is due to rounding errors. Every computer ultimately stores and manipulate numbers using floating points arithmetic [13]. Equations (7) and (8) involve the subtraction of two numbers that are very close to each other, which leads to large cancellation errors for small values of ε than are amplified by the division by ε. On the other hand, large values of the stepsize give inaccurate estimations of the gradient. Finding the optimal value of ε that trade-offs these two effects is sometimes called the *stepsize dilemma* [14]. Due to this, some heuristics and algorithms had been introduced in order to pick the value of ε that will minimize the error [15, 14], but all these methods require some a priori knowledge about the function to be differentiated. If well many analytical functions, like polynomials and trigonometric functions, can be computed with machine precision, numerical solutions of differential equations have errors larger than machine precision, which leads to inaccurate estimations of the gradient when ε is too small. We will further emphatize this point in Section 4.

Replacing derivatives by finite differences is also a common practice when solving partial differential equations (PDEs), a technique known as the *method of lines* [16]. To illustrate this point, let us consider the case of the one-dimensional heat equation

$$\frac{\partial u}{\partial t} = D\,\frac{\partial^2 u}{\partial x^2}, \quad u(0,t) = \alpha(t), \quad u(1,t) = \beta(t) \tag{9}$$

that includes both spatial and temporal partial derivatives of the unknown function $u(x,t)$. In order to numerically solve this equation, we can define a spatial grid with coordinates $m\Delta x$, $m = 0, 1, 2, \ldots, N$ and $\Delta x = 1/N$. If we call $u_m(t) = u(m\Delta x, t)$ the value of the solution evaluated in the fixed points in the grid, then we can replace the second order partial derivative in Equation (9) by the corresponding second order finite difference[1]

$$\frac{du_m}{dt} = D\frac{u_{m-1} - 2u_m + u_{m+1}}{\Delta x^2} \tag{10}$$

for $m = 1, 2, \ldots, N-1$ (in the extremes we simply have $u_0(t) = \alpha(t)$ and $u_N(t) = \beta(t)$). Now, equation (10) is a system of ordinary differential equations (just temporal derivatives) with a total of $N-1$ equations. This can be solved directly using an ODE solver. Further improvements can be made by exploiting the fact that the coupling between the different functions u_m is sparse, that is, the temporal derivative of u_m just depends of the values of the function in the neighbour points in the grid.

3.3 Complex step differentiation

An alternative to finite differences that avoids rounding errors is based complex variable analysis. The first proposals originated in 1967 using the Cauchy integral theorem involving the numerical evaluation of a complex valued integral [17, 18]. A new approach recently emerged that uses the Taylor expansion of a function to define its complex generalization [19, 20]. Assuming that we have one single scalar parameter $\theta \in \mathbb{R}$, then the function $L(\theta)$

[1]Since $u_m(t)$ is a function of one single variable, we write the total derivative $\frac{du_m}{dt}$ instead of the partial derivative symbol used before $\frac{\partial u}{\partial t}$, which it is usually used just for multivariable function.

can be expanded as the Taylor expansion

$$L(\theta + i\varepsilon) = L(\theta) + i\varepsilon L'(\theta) - \frac{1}{2}L''(\theta)\varepsilon^2 + \mathcal{O}(\varepsilon^3), \tag{11}$$

where i is the imaginary unit satisfying $i^2 = -1$. From this equation we can observed that many factors vanish when we compute the imaginary part $\mathrm{Im}(L(\theta + i\varepsilon))$, which leads to

$$L'(\theta) = \frac{\mathrm{Im}(L(\theta + i\varepsilon))}{\varepsilon} + \mathcal{O}(\varepsilon^2) \tag{12}$$

The method of *complex step differentiation* consist then in estimating the gradient as $\mathrm{Im}(L(\theta + i\varepsilon))/\varepsilon$ for a small value of ε. Besides the advantage of being a method with precision $\mathcal{O}(\varepsilon^2)$, the complex step method avoids subtracting cancellation error and then the value of ε can be reduce to almost machine precision error without affecting the calculation of the derivative.

Extension to higher order derivatives can be done by introducing multicomplex variables [21].

3.4 Automatic differentiation

Automatic differentiation (AD) is a technology that allows computing gradients thought a computer program [22]. The main idea is that every computer program manipulating numbers can be reduced to a sequence of simple algebraic operations that can be easily differentiable. The derivatives of the outputs of the computer program with respect to their inputs are then combined using the chain rule. One advantage of AD systems is that we can automatically differentiate programs that include control flow, such as branching, loops or recursions. This is because at the end of the day, any program can be reduced to a trace of input, intermediate and output variables [23].

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

3.4.1 Forward mode

Forward mode AD can be implemented in different ways depending on the data structures we use at the moment of representing a computer program. Examples of these data structures include dual numbers and Wengert lists (see [23] for a good review on these methods).

Dual numbers

Let us first consider the case of dual numbers. The idea is to extend the definition of a numerical variable that takes a certain value to also carry information about its derivative with respect to certain scalar parameter $\theta \in \mathbb{R}$. We can define an abstract type, defined as a dual number, composed of two elements: a *value* coordinate x_1 that carries the value of the variable and a *derivative* coordinate x_2 with the value of the derivative $\frac{\partial x_1}{\partial \theta}$. Just as complex number, we can represent dual numbers in the vectorial form (x_1, x_2) or in the rectangular form

$$x_\epsilon = x_1 + \epsilon\, x_2 \tag{13}$$

where ϵ is an abstract number 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. 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

$$x_\epsilon + y_\epsilon = (x_1 + y_1) + \epsilon\,(x_2 + y_2) \qquad x_\epsilon y_\epsilon = x_1 y_1 + \epsilon\,(x_1 y_2 + x_2 y_1). \tag{14}$$

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

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

$$\begin{aligned} f(x_1 + \epsilon x_2) &= f(x_1) + \epsilon\, x_2\, f'(x_1) + \epsilon^2 \cdot (\ldots) \\ &= f(x_1) + \epsilon\, x_2\, f'(x_1) \end{aligned} \tag{15}$$

When computing first order derivatives, we can ignore everything of order ϵ^2 or larger, which is represented in the condition $\epsilon^2 = 0$. This implies that we can use dual numbers to implement forward AD through a numerical algorithm. We will explore how this is carried in Section 4.

Computational graph

An useful way of representing a computer program is via a computational graph with intermediate variables that relate the input and output variables. Most scalar functions of interest can be represented in this factorial form as a acyclic directed graph with nodes associated to variables and edges to atomic operations [22, 24], known as Kantorovich graph [25]. We can define $v_1, v_2, \ldots, v_p = \theta_1, \theta_2, \ldots, \theta_p$ the input set of variables; $v_{p+1}, \ldots, v_{m-1}$ the set of all the intermediate variables, and finally $v_m = L(\theta)$ the final output of a computer program. This can be done in such a way that the order is strict, meaning that each variable v_i is computed just as a function of the previous variables v_j with $j < i$. Once the graph is constructed, we can compute the derivative of every node with respect to other using Bauer formula [26, 27]

$$\frac{\partial v_j}{\partial v_i} = \sum_{\substack{\text{paths } w_0 \to w_1 \to \ldots \to w_K \\ \text{with } w_0 = v_i, w_K = v_j}} \prod_{k=0}^{K-1} \frac{\partial w_{k+1}}{\partial w_k}, \tag{16}$$

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

$$\frac{\partial v_j}{\partial v_i} = \sum_{w \text{ such that } w \to v_j} \frac{\partial v_j}{\partial w} \frac{\partial w}{\partial v_i} \tag{17}$$

Since every variable node w such that $w \to v_j$ is an edge of the computational graph have index less than j, we can iterate this procedure as we run the computer program and solve for both the function and its gradient. This is possible because in forward mode the term $\frac{\partial w}{\partial v_i}$ has been computed in a previous iteration, while $\frac{\partial v_j}{\partial w}$ can be evaluated at the same time the node v_j is computed based on only the value of the parent variable nodes.

3.4.2 Backward mode

Backward mode AD is also known as the adjoint of cotangent linear mode, or backpropagation in the field of machine learning. The reverse mode of automatic differentiation has been introduced in different contexts [28] and materializes the observation made by Phil Wolfe that if the chain rule is implemented in reverse mode, then the ratio between the computation of the gradient of a function and the function itself can be bounded by a constant that do not depend of the number of parameters to differentiate [24, 29], a point known as *cheap gradient principle* [28]. Given a directional graph of operations defined by a Wengert list [30], we can compute gradients of any given function in the same fashion as Equation (17) but in backwards mode as

$$\bar{v}_i = \frac{\partial \ell}{\partial v_i} = \sum_{w:v\to w\in G} \frac{\partial w}{\partial v}\bar{w}. \tag{18}$$

Here we have introduce the notation $\bar{\omega} = \frac{\partial \ell}{\partial \omega}$ to indicate that the partial derivative is always of the final loss function with respect to the different program variables. Since in backwards AD the values of $\bar{\omega}$ are being updated in reverse order, in order to evaluate the terms $\frac{\partial \omega}{\partial v}$ we need to know the state value of all the argument variables v of ω, which need to be stored in memory during the evaluation of the function in order to be able to apply backward AD.

Another way of implementing backwards AD is by defining a *pullback* function [31], a method also known as *continuation-passing style* [32]. In the backward step, this executes a series of functions calls, one for each elementary operation. If one of the nodes in the graph w is the output of an operation involving the nodes $v_1, \ldots, v_m$, where $v_i \to w$ are all nodes in the graph, then the pullback $\bar{v}_1, \ldots, \bar{v}_m = \mathcal{B}_w(\bar{w})$ is a function that accepts gradients with respect to w (defined as $\bar{w}$) and returns gradients with respect to each v_i ($\bar{v}_i$) by applying the chain rule. Consider the example of the multiplicative operation $w = v_1 \times v_2$. Then

$$\bar{v}_1, \bar{v}_2 = v_2 \times \bar{w}, \quad v_1 \times \bar{w} = \mathcal{B}_w(\bar{w}), \tag{19}$$

which is equivalent to using the chain rule as

$$\frac{\partial \ell}{\partial v_1} = \frac{\partial}{\partial v_1}(v_1 \times v_2)\frac{\partial \ell}{\partial w}. \tag{20}$$

3.4.3 AD connection with JVPs and VJPs

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

For nested functions, the Jacobian is described as the product of multiple Jacobian using the chain rule. In this case, the full gradient is computed as the chain product of vectors and Jacobians. Let us consider for example the case of a loss function $L : \mathbb{R}^p \mapsto \mathbb{R}$ that can be decomposed as $L(\theta) = \ell \circ g_k \circ \ldots \circ g_2 \circ g_1(\theta)$, with $\ell : \mathbb{R}^{d_k} \mapsto \mathbb{R}$ the final evaluation of the

loss function after we apply in order a sequence of intermediate functions $g_i : \mathbb{R}^{d_{i-1}} \mapsto \mathbb{R}^{d_i}$, $d_0 = p$. Now, using the chain rule, we can calculate the gradient of the final loss function as

$$\nabla_\theta L = \nabla \ell \cdot Dg_k \cdot Dg_{k-1} \cdot \ldots \cdot Dg_2 \cdot Dg_1, \tag{21}$$

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

$$d_2 d_1 n + d_3 d_2 p + \ldots + d_k d_{k-1} p + d_k p = \mathcal{O}(p) \tag{22}$$

operations, while backwards mode AD requires

$$d_k d_{k-1} + d_{k-1} d_{k-2} + \ldots + d_2 d_1 + d_1 n = \mathcal{O}(1) \tag{23}$$

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

When the function to differentiate has a larger input space than output, AD in backward mode is more efficient as it propagates the chain rule by computing VJPs, reason why backwards AD is more used in modern machine learning. However, notice that backwards mode AD requires us to save the solution thought the forward run in order to run backwards afterwards [33], while in forward mode we can just evaluate the gradient as we iterate our sequence of functions. This means that for problems with a small number of parameters, forward mode can be faster and more memory-efficient that backwards AD.

3.5 Symbolic differentiation

Sometimes AD is compared against symbolic differentiation. According to [34], these two are the same and the only difference is in the data structures used to implement them, while [35] suggests that AD is symbolic differentiation performed by a compiler.

3.6 Sensitivity equations

An easy way to derive an expression for the sensitivity s is by deriving the sensitivity equations [4], a method also referred to as continuous local sensitivity analysis (CSA). If we consider the original system of ODEs and we differentiate with respect to θ, we then obtain

$$\frac{d}{d\theta}\frac{du}{dt} = \frac{d}{d\theta} f(u(\theta), \theta, t) = \frac{\partial f}{\partial \theta} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial \theta}, \tag{24}$$

that is

$$\frac{ds}{dt} = \frac{\partial f}{\partial u} s + \frac{\partial f}{\partial \theta}. \tag{25}$$

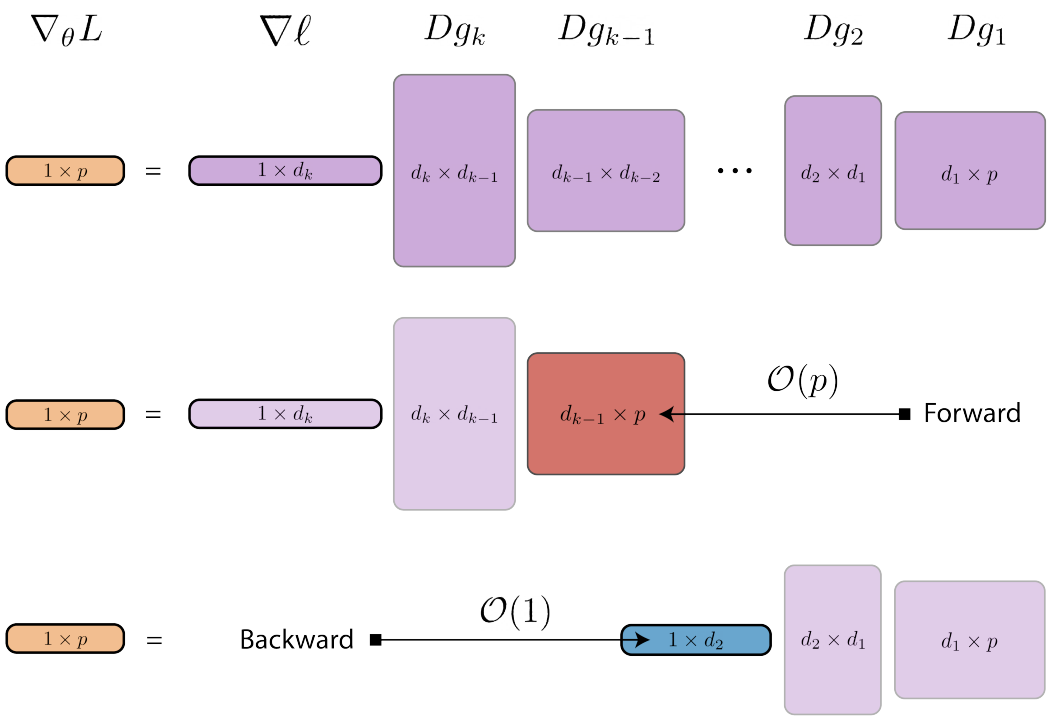

Figure 2: Comparison between forward and backward AD. Changing the order how we multiply the Jacobians change the total number of floating-point operations, which leads to different computational complexities between forward and backward mode. However, backwards mode requires to store in memory information about the forward execution of the program, while forward mode can update the gradient on running time.

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

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

For systems of equations with few number of parameters, this method is useful since the system of equations composed by Equations (1) and (25) can be solved in $\mathcal{O}(np)$ using the same precision for both solution and sensitivity numerical evaluation. Furthermore, this method does not required saving the solution in memory, so it can be solved purely in forward mode without backtracking operations.

3.7 Adjoint methods

3.7.1 Discrete adjoint method

Also know as the adjoint state method, it is another example of a discrete method that aims to find the gradient by solving an alternative system of linear equations, known as the *adjoint equations*, at the same time that we solve the original system of linear 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 [36, 5]. This approach follows the discretize-optimize approach, meaning that we first discretize the system of continuous ODEs and then solve on top of these linear equations [5]. Just as in the case of automatic differentiation, the set of adjoint equations can be solved in both forward and backward mode.

Discrete differential equation

The first step in order to derive the adjoint equation is to discretize the set of differential equations in (1) into finite evaluations of the function $u(t;\theta)$. Given the sequence of timesteps $t_0, t_1, \ldots, t_N$, we evaluate the solution at $u_i = u(t_i;\theta)$. In the case of using an explicit numerical solver, these values will be constrained to satisfy a set of equations of the form

$$u_{i+1} = A_i(\theta)\, u_i + b_i \tag{26}$$

with $A_i \in \mathbb{R}^{n\times n}$ a squared matrix defined by the numerical solver. Solving the differential equation then implies to be able to solve the system of constraints

$$g_i(u_{i+1};\theta) = u_{i+1} - A_i(\theta)\, u_i - b_i = 0 \tag{27}$$

for all $i = 0, 1, \ldots, N-1$. For most cases, this system can be solved sequentially, by solving for u_i in increasing order of index. If we call the super-vector $U = (u_1, u_2, \ldots, u_N) \in \mathbb{R}^{nN}$,

we can combine all these equations in into one single system of linear equations

$$A(\theta)U = \begin{bmatrix} \mathbb{I}_{n\times n} & 0 & & & \\ -A_1 & \mathbb{I}_{n\times n} & 0 & & \\ & -A_2 & \mathbb{I}_{n\times n} & 0 & \\ & & & \ddots & \\ & & & -A_{N-1} & \mathbb{I}_{n\times n} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} A_0 u_0 + b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-1} \end{bmatrix} = b(\theta), \tag{28}$$

with $\mathbb{I}_{n\times n}$ the identity matrix of size $n \times n$. It is usually convenient to write this system of linear equations in the residual form $G(U;\theta) = 0$, where $G(U;\theta) = A(\theta)U - b(\theta)$ is the residual between both sides of the equation. Different numerical schemes will lead to different design matrix $A(\theta)$ and vector $b(\theta)$, but ultimately every numerical method will lead to a system of linear equations with the form $G(U;\theta) = A(\theta)U - b(\theta) = 0$ after being discretized. It is important to notice that in most cases, the matrix $A(\theta)$ is quite large and mostly sparse. If well this representation of the discrete differential equation is quite convenient for mathematical manipulations, at the moment of solving the system we will rely in iterative solvers that save memory and computation.

Adjoint state equations

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

$$L(U,\theta) = \frac{1}{2}\sum_{i=1}^{N} \|u_i - u_i^{\text{obs}}\|^2, \tag{29}$$

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

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial \theta}, \tag{30}$$

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

$$\frac{dG}{d\theta} = \frac{\partial G}{\partial \theta} + \frac{\partial G}{\partial U}\frac{\partial U}{\partial \theta} = 0 \tag{31}$$

which is equivalent to

$$\frac{\partial U}{\partial \theta} = -\left(\frac{\partial G}{\partial U}\right)^{-1}\frac{\partial G}{\partial \theta}. \tag{32}$$

If we replace this last expression into equation (30), we obtain

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \underbrace{\frac{\partial L}{\partial U}}_{\text{vector}}\left(\frac{\partial G}{\partial U}\right)^{-1}\frac{\partial G}{\partial \theta}. \tag{33}$$

The important trick in the adjoint state methods is to observe that in this last equation, the right-hand side can be resolved as a vector-Jacobian product (VJP). Instead of computing

the product of the matrices $\left(\frac{\partial G}{\partial U}\right)^{-1}$ and $\frac{\partial G}{\partial \theta}$, it is computationally more efficient first to compute the resulting vector from the operation $\frac{\partial L}{\partial U}\left(\frac{\partial G}{\partial U}\right)^{-1}$ and then multiply this by $\frac{\partial G}{\partial \theta}$. This is what leads to the definition of the adjoint $\lambda \in \mathbb{R}^{nN}$ as the solution of the linear system of equations

$$\left(\frac{\partial G}{\partial U}\right)^T \lambda = \left(\frac{\partial L}{\partial U}\right)^T, \tag{34}$$

that is,

$$\lambda^T = \frac{\partial L}{\partial U}\left(\frac{\partial g}{\partial U}\right)^{-1}. \tag{35}$$

Finally, if we replace Equation (35) into (33), we obtain

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \lambda^T \frac{\partial G}{\partial \theta}. \tag{36}$$

The important trick to notice here is the rearrangement of the multiplicative terms involved in equation (33). Computing the full Jacobian/sensitivity $\partial u/\partial \theta$ will be computationally expensive and involves the product of two matrices. However, we are not interested in the calculation of the Jacobian, but instead in the VJP given by $\frac{\partial L}{\partial U}\frac{\partial U}{\partial \theta}$. By rearranging these terms, we can make the same computation more efficient.

For the linear system of discrete equations $G(U;\theta) = 0$, we have [37]

$$\frac{\partial G}{\partial \theta} = \frac{\partial A}{\partial \theta}U - \frac{\partial b}{\partial \theta}, \tag{37}$$

so the desired gradient in Equation (36) can be computed as

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \lambda^T\left(\frac{\partial A}{\partial \theta}U - \frac{\partial b}{\partial \theta}\right) \tag{38}$$

with λ the solution of the linear system (Equation (34))

$$A(\theta)^T\lambda = \begin{bmatrix} \mathbb{I}_{n\times n} & -A_1^T & & & \\ 0 & \mathbb{I}_{n\times n} & -A_2^T & & \\ & 0 & \mathbb{I}_{n\times n} & -A_3^T & \\ & & & \ddots & -A_{N-1}^T \\ & & & 0 & \mathbb{I}_{n\times n} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \vdots \\ \lambda_N \end{bmatrix} = \begin{bmatrix} u_1 - u_1^{\text{obs}} \\ u_2 - u_2^{\text{obs}} \\ u_3 - u_3^{\text{obs}} \\ \vdots \\ u_N - u_N^{\text{obs}} \end{bmatrix} = \frac{\partial L}{\partial U}^T. \tag{39}$$

This is a linear system of equations with the same size of the original $A(\theta)U = b(\theta)$, but involving the adjoint matrix A^T. Computationally this also means that if we can solve the original system of discretized equations then we can also solve the adjoint. One way of doing this is relying on matrix factorization. Using the LU factorization we can write the matrix $A(\theta)$ as the product of a lower and upper triangular matrices $A(\theta) = LU$, which then can be also used for solving the adjoint equation since $A^T(\theta) = U^T L^T$. Another more natural way of finding the adjoins λ is by noticing that the system of equations (39) is equivalent to the iterative scheme

$$\lambda_i = A_i^T\lambda_{i+1} + (u_i - u_i^{\text{obs}}) \tag{40}$$

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

3.7.2 Continuous adjoint method

The continuous adjoint method, also known as continuous adjoint sensitivity analysis (CASA), operates by defining a convenient set of new differential equations for the adjoint variable and using this to compute the gradient in a more efficient manner. Mathematically speaking, the adjoint equations can be derived from a duality or Lagrangian point of view [5]. We prefer to derive the equation using the former methods since we believe it gives better insights to how the method works and also allow to generalize to other user cases. The derivation of both the discrete and continuous adjoint methods using Lagrangian multipliers can be found in Appendix A. We encourage the interested reader to make the effort of following how the continuous adjoint method follows the same logic than the discrete methods, but where the discretization of the differential equation does not happen until the very last step, when the solutions of the differential equations involved need to be numerically evaluated.

Consider an integrated loss function of the form

$$L(u;\theta) = \int_{\mathrm{t}_0}^{\mathrm{t}_1} h(u(t;\theta),\theta)dt \tag{41}$$

and its derivative with respect to the parameter θ given by

$$\frac{dL}{d\theta} = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(\frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u}s(t)\right) dt. \tag{42}$$

As explained in previous section, the complicated term to evaluate in the last expression is the sensitivity (Equation (6)). Just as in the case of the discrete adjoint method, the trick is to evaluate the VJP $\frac{\partial h}{\partial u}\frac{\partial u}{\partial \theta}$ by defining an intermediate adjoint variable. The continuous adjoint equation now is obtained by finding the dual/adjoint equation of the sensitivity equation using the weak formulation of Equation (25). The adjoint equation is obtained by writing the sensitivity equation in the form

$$\int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \left(\frac{ds}{dt} - f(u,\theta,t)\, s - \frac{\partial f}{\partial \theta}\right) dt = 0, \tag{43}$$

where this equation must be satisfied for every function $\lambda(t)$ in order to Equation (55) to be true. The next step is to get rid of all time derivative applied to the sensitivity $s(t)$ using integration by parts:

$$\int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \frac{ds}{dt}dt = \lambda(t_1)^T s(t_1) - \lambda(t_0)^T s(t_0) - \int_{\mathrm{t}_0}^{\mathrm{t}_1} \frac{d\lambda^T}{dt} s(t)\, dt. \tag{44}$$

Replacing this last expression into Equation (43) we obtain

$$\int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(-\frac{d\lambda^T}{dt} - \lambda(t)^T f(u,\theta,t)\right) s(t)dt = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \frac{\partial f}{\partial \theta}dt - \lambda(t_1)^T s(t_1) + \lambda(t_0)^T s(t_0). \tag{45}$$

At first glance, there is nothing particularly interesting about this last equation. However, both Equations (42) and (45) involved a VJP with $s(t)$. Since Equation (45) must hold for every function $\lambda(t)$, we can pick $\lambda(t)$ to make the terms involving $s(t)$ in Equations (42) and (45) to perfectly coincide. This is done by defining the adjoint $\lambda(t)$ to be the solution of the new system of differential equations

$$\frac{d\lambda}{dt} = -f(u,\theta,t)^T\lambda - \frac{\partial h^T}{\partial u} \qquad \lambda(t_1) = 0. \tag{46}$$

Notice that the adjoint equation is define with 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 (45) to

$$\int_{t_0}^{t_1} \frac{\partial h}{\partial u} s(t)dt = \lambda(t_0)^T s(t_0) + \int_{t_0}^{t_1} \lambda^T(t)\frac{\partial f}{\partial \theta}dt. \tag{47}$$

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

$$\frac{dL}{d\theta} = \lambda^T(t_0)s(t_0) + \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T\frac{\partial f}{\partial \theta}\right) dt \tag{48}$$

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

1. Solve the original differential equation $\frac{du}{dt} = f(u,t,\theta)$;
2. Solve the backwards adjoint differential equation (46);
3. Compute the gradient using Equation (48).

4 Computational implementation

In this section we are going to address how these tools are computationally implemented and how to decide which method to use depending the problem.

4.1 Forward discrete methods

Let us start comparing how finite differences and complex-step differentiation perform for a simple problem with one single parameter. Figure 3 illustrates the error in computing the gradient of a simple loss function for both true analytical solution and numerical solution of a system of ODEs as a function of the stepsize ε.

In general, finite differences is very easy to implement since it does not require any type of software support, but it is a less accurate and as costly as forward AD [24] and comple-step differentiation. Implementing these last types in a programming language implies defining what it means to perform basic operations and then combine them using the chain rule provided by the dual number properties. In Julia we can create a dual number by defining an object with two values, and then extend the definition of algebraic operations and functions, a process known as *operator overloading* [39], by relying in multiple dispatch:

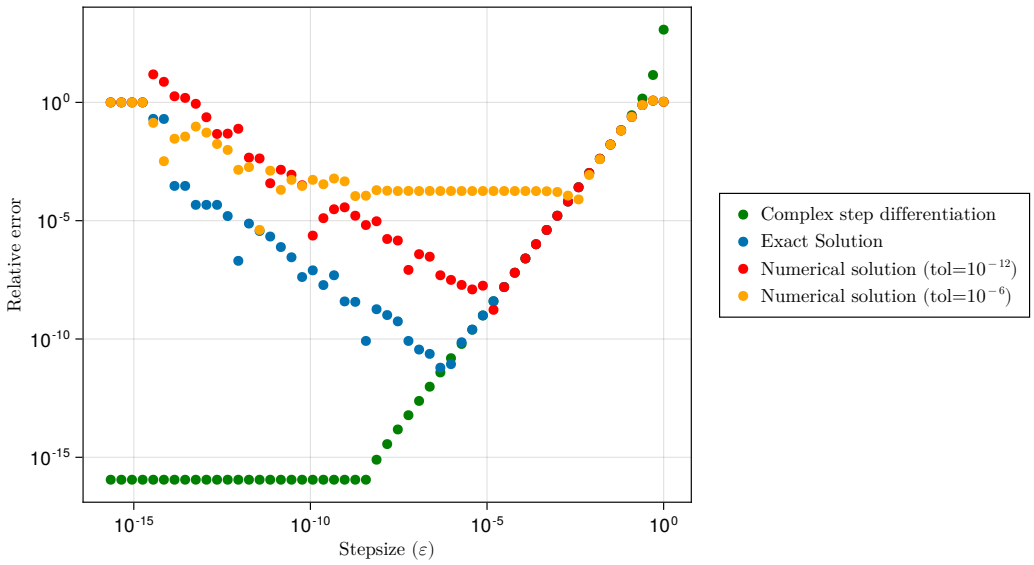

Figure 3: Absolute relative error when computing the gradient of the function $u(t) = \sin(\omega t)/\omega$ with respect to ω at $t = 10.0$ as a function of the stepsize ε. Here $u(t)$ corresponds to the solution of the differential equation $u'' + \omega^2 u = 0$ with initial condition $u(0) = 0$ and $u'(0) = 1$. The blue dots correspond to the case where this is computed finite differences. The red and orange lines are for the case where $u(t)$ is numerically computed using the default Tsitouras solver [38] from `OrdinaryDiffEq.jl` using different tolerances. The error when using a numerical solver is larger and it is dependent of the numerical precision of the numerical solver.

```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)
```

and then we can simply evaluate derivatives by evaluating the derivative component of the dual number that results from the combination of operations

```julia
a = DualNumber(value=1.0, derivative=1.0)

b = DualNumber(value=2.0, derivative=0.0)
c = DualNumber(value=3.0, derivative=0.0)

result = a * b * c
println("The derivative of a*b*c with respect to a is: ", result.derivative)
```

Notice that in this last example the dual numbers `b` and `c` where initialized with derivative value equals to zero, while `a` with value equals to one. This is because we were interested in computing the derivative with respect to `a`, and then $\frac{\partial a}{\partial a} = 1$, while $\frac{\partial b}{\partial a} = \frac{\partial c}{\partial a} = 0$.

We can also extend the definition of standard functions by simply applying the chain rule and storing the derivative in the dual variable following Equation (15):

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

With all these pieces together, we are able to propagate forward the value of a single-valued derivative though a series of algebraic operations.

In the Julia ecosystem, `ForwardDiff.jl` implements forward mode AD with multidimensional dual numbers [40]. Notice that a major limitation of the dual number approach is that we need a dual variable for each variable we want to differentiate. Implementations of forward AD using dual numbers and computational graphs require a number of operations that increases with the number of variables to differentiate, since each computed quantity is accompanied by the corresponding gradient calculations [24]. This consideration applies to most forward methods.

Notice that both AD based in dual number and complex-step differentiation are based on defining and extended variable that carries information about the gradient. Both methods introduce an abstract unit (ϵ and i, respectively) associated to the imaginary part of the extender value that carries forward the numerical value of the gradient. This resemblance between the methods makes them to be susceptible to the same advantages and disadvantages: easiness to implement with operator overloading; inefficient scaling with respect to the number of variables to differentiate. However, although these methods seems very similar, it is important to remark that AD gives the exact gradient, while complex step differentiation relies in numerical approximations that are valid just when the stepsize ε is small. The next example shows how the calculation of the gradient of $\sin(x^2)$ is performed by these two methods:

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

(49)

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

$$\lim_{\varepsilon \to 0} \frac{1}{\varepsilon}\,\cos(x^2 - \varepsilon^2)\sinh(2i\varepsilon) = \cos(x^2)(2x). \tag{50}$$

The stepsize dependence of the complex step differentiation method makes it resemble more

to finite differences than AD with dual numbers. This difference between the methods also makes the complex step method sometimes more efficient than both finite differences and AD [21], an effect that can be counterbalanced by the number of extra unnecessary operation that complex arithmetic requires (see last column of (49)) [20].

4.2 Backwards methods

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

4.3 Solving the adjoint

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

(i) Solve for $u(t)$ again backwards.

(ii) Store $u(t)$ in memory during the forward step.

(iii) Store reference values in memory and interpolate in between. This technique is known as *checkpoiting* or windowing that tradeoffs computational running time and storage. This is implemented in `Checkpoiting.jl` [41].

Computing the ODE backwards can be unstable and lead to exponential errors [42]. In [43], the solution is recalculated backwards together with the adjoint simulating an augmented dynamics:

$$\frac{d}{dt}\begin{bmatrix} u \\ \lambda \\ \frac{dL}{d\theta} \end{bmatrix} = \begin{bmatrix} -f \\ -\lambda^T \frac{\partial f}{\partial u} \\ -\lambda^T \frac{\partial f}{\partial \theta} \end{bmatrix} = -[1, \lambda^T, \lambda^T] \begin{bmatrix} f & \frac{\partial f}{\partial u} & \frac{\partial f}{\partial \theta} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{51}$$

with initial condition $[u(t_1), \frac{\partial L}{\partial u(t_1)}, 0]$. One way of solving this system of equations that ensures stability is by using implicit methods. However, this requires cubic time in the total number of ordinary differential equations, leading to a total complexity of $\mathcal{O}((n+p)^3)$ for the adjoint method. Two alternatives are proposed in [42], the first one called *Quadrature Adjoint* produces a high order interpolation of the solution $u(t)$ as we move forward, then solve for λ backwards using an implicit solver and finally integrating $\frac{dL}{d\theta}$ in a forward step. This reduces the complexity to $\mathcal{O}(n^3+p)$, where the cubic cost in the number of ODEs comes from the fact that we still need to solve the original stiff differential equation in the forward step. A second but similar approach is to use a implicit-explicit (IMEX) solver, where we use the implicit part for the original equation and the explicit for the adjoint. This method also will have complexity $\mathcal{O}(n^3 + p)$.

Appendices

A Lagrangian derivation of adjoints

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

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

$$I(\theta) = L(\theta) - \int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \left(\frac{du}{dt} - f(u, \theta, t) \right) dt \tag{52}$$

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

$$\frac{dL}{d\theta} = \frac{dI}{d\theta} = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \left(\frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u}\frac{\partial u}{\partial \theta} \right) dt - \int_{\mathrm{t}_0}^{\mathrm{t}_1} \lambda(t)^T \left(\frac{d}{dt}\frac{du}{d\theta} - \frac{\partial f}{\partial u}\frac{du}{d\theta} - \frac{\partial f}{\partial \theta} \right) dt. \tag{53}$$

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

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

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

$$\frac{d\lambda}{d\theta} = -\left(\frac{\partial f}{\partial u} \right)^T \lambda - \left(\frac{\partial h}{\partial u} \right)^T, \tag{55}$$

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

References

[1] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[2] Bharath Ramsundar, Dilip Krishnamurthy, and Venkatasubramanian Viswanathan. "Differentiable Physics: A Position Piece". In: *arXiv* (2021). DOI: 10.48550/arxiv.2109.07573.

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

[4] James Ramsay and Giles Hooker. *Dynamic data analysis*. Springer, 2017.

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

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

[7] Christopher Rackauckas et al. "Universal differential equations for scientific machine learning". In: *arXiv preprint arXiv:2001.04385* (2020).

[8] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. ISSN: 0036-1445. DOI: 10.1137/141000671.

[9] Andrew M Bradley. *PDE-constrained optimization and the adjoint method*. Tech. rep. Technical Report. Stanford University. https://cs. stanford. edu/~ ambrad ..., 2013.

[10] Derek Onken and Lars Ruthotto. "Discretize-Optimize vs. Optimize-Discretize for Time-Series Regression and Continuous Normalizing Flows". In: *arXiv* (2020). DOI: 10.48550/arxiv.2005.13420.

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

[12] Uri M. Ascher and Chen Greif. *A First Course in Numerical Methods*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/9780898719987. eprint: https://epubs.siam.org/doi/pdf/10.1137/9780898719987. URL: https://epubs.siam.org/doi/abs/10.1137/9780898719987.

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

[14] Ravishankar Mathur. "An analytical approach to computing step sizes for finite-difference derivatives". PhD thesis. 2012.

[15] Russell R. Barton. "Computing Forward Difference Derivatives In Engineering Optimization". In: *Engineering Optimization* 20.3 (1992), pp. 205–224. ISSN: 0305-215X. DOI: 10.1080/03052159208941281.

[16] Uri M Ascher. *Numerical methods for evolutionary differential equations*. SIAM, 2008.

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

[18] J N Lyness and C B Moler. "Numerical Differentiation of Analytic Functions". In: *SIAM Journal on Numerical Analysis* 4.2 (1967), pp. 202–210. ISSN: 0036-1429. DOI: 10.1137/0704019.

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

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

[21] Gregory Lantoine, Ryan P. Russell, and Thierry Dargent. "Using Multicomplex Variables for Automatic Computation of High-Order Derivatives". In: *ACM Transactions on Mathematical Software (TOMS)* 38.3 (2012), p. 16. ISSN: 0098-3500. DOI: 10.1145/2168773.2168774.

[22] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[23] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: *arXiv* (2015). DOI: 10.48550/arxiv.1502.05767.

[24] Andreas Griewank. "On Automatic Differentiation". In: (Feb. 1997).

[25] Leonid Vitalevich Kantorovich. "On a mathematical symbolism convenient for performing machine calculations". In: *Dokl. Akad. Nauk SSSR*. Vol. 113. 4. 1957, pp. 738–741.

[26] F L Bauer. "Computational Graphs and Rounding Error". In: *SIAM Journal on Numerical Analysis* 11.1 (1974), pp. 87–96. ISSN: 0036-1429. DOI: 10.1137/0711010.

[27] Deniz Oktay et al. "Randomized Automatic Differentiation". In: *arXiv* (2020). DOI: 10.48550/arxiv.2007.10412.

[28] Andreas Griewank. "Who invented the reverse mode of differentiation". In: *Documenta Mathematica, Extra Volume ISMP* 389400 (2012).

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

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

[31] Michael Innes. "Don't Unroll Adjoint: Differentiating SSA-Form Programs". In: *arXiv* (2018).

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

[33] C H Bennett. "Logical Reversibility of Computation". In: *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532. ISSN: 0018-8646. DOI: 10.1147/rd.176.0525.

[34] Soeren Laue. *On the Equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation*. 2019. DOI: 10.48550/ARXIV.1904.02990. URL: https://arxiv.org/abs/1904.02990.

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

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

[37] Steven G. Johnson. "Notes on Adjoint Methods for 18.335". In: 2012.

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

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

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

[41] Michel Schanen et al. "Transparent Checkpointing for Automatic Differentiation of Program Loops Through Expression Transformations". In: *Lecture Notes in Computer Science* (2023), pp. 483–497. ISSN: 0302-9743. DOI: 10.1007/978-3-031-36024-4_37.

[42] Suyong Kim et al. "Stiff neural ordinary differential equations". en. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 31.9 (Sept. 2021), p. 093122. ISSN: 1054-1500, 1089-7682. DOI: 10.1063/5.0060697. URL: https://aip.scitation.org/doi/10.1063/5.0060697 (visited on 02/25/2022).

[43] Ricky T. Q. Chen et al. "Neural Ordinary Differential Equations". In: *arXiv:1806.07366 [cs, stat]* (Dec. 2019). arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366 (visited on 02/25/2022).