The Adjoint Method for ODEs

Claus Sarnighausen[1] and Constantin Scheffold[1]

[1]*Department of Physics and Astronomy, University of Heidelberg, 69117 Heidelberg, Germany*

(Dated: June 30, 2023)

Abstract: This write-up focuses on the mathematics underlying packages used by machine learning scientists, which are essential for understanding the behavior and problems of machine learning models. The concept of automatic differentiation (AD) is introduced as a method for calculating derivatives by dissecting a program into primitive operations and applying the chain rule. Dual numbers are presented as a tool for efficient evaluation of function values and derivatives in one forward function call. Adjoint models, which efficiently calculate total derivatives of outputs with respect to parameters or inputs, are discussed. The process of constructing adjoint models for different scenarios is explained, such as for solving nonlinear systems and the adjoint state method for ordinary differential equations.

I. INTRODUCTION

Many supervised Machine Learning (ML) and physics models consist of input data x_{input}, a mapping process producing outputs $f_\theta(x_{input}) = y$ and a loss function which compares the output with known output data. In many algorithms, the "learning" (optimization) involves calculating the influence of a single parameter on the Loss function $\frac{dL}{d\theta}$ and changing them accordingly.

The need for efficient and precise computation of large system derivatives has led to the development of automatic differentiation (AD). We will introduce the fundamentals of AD with the adjoint method, where the popular technique of backpropagation turns out to be only a special case. With the help of the adjoint method we will learn to optimise models in different problem settings. Buiding up from explicit and implicit models to models exploiting ODEs and their rich numerical methods, the so called neural ordinary differential equations. In the outlook, we go back to a recent development in machine learning that uses this technique: neuralODEs. As often in SciML, techniques from Machine learning are applied to classical scientific computing problems.

In this write-up, we present the mathematics underlying packages an ML scientist uses. They may not be necessary to implement ML models, but inevitable to understand their behavior and problems.

II. AUTOMATIC DIFFERENTIATION

A. Chain rule on primitives

In automatic differentiation (AD), derivatives are calculated by dissecting a program into primitive operations (primitives) and then applying the chain rule. This method is as precise as symbolic differentiation since it evaluates the derivatives exactly and requires less computational resources than numeric differentiation because a partial can be evaluated in one function sweep. [1].

Consider a model in the form of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$. The key idea here is that the function can be dissected into primitive functions with known differentiation rules.

$$f = f^L \circ f^{L-1} \circ \ldots \circ f^1 \tag{1}$$

The partial derivative of f with respect to its input vector $x \in R^n$ is the jacobian matrix $J_{ij} = \frac{\partial f_i}{\partial x_j}$, which can be evaluated using the chain rule.

$$J = J_L J_{L-1} \ldots J_1 \tag{2}$$

B. Forward vs. Backward

In *forward mode AD*, the jacobi matrices are multiplied together with the forward pass - from low to high index or from right to left in Equation 2. Conversely, *reverse mode AD* starts with the forward pass, stores the values and then performs the matrix multiplications from high to low index or from left to right in Equation 2. Performing operations on one unit vector x_i in forward mode or one output component $f_i(\mathbf{x})$ in backward mode is called performing one *sweep*. In forward mode AD, each sweep calculates one column of J (directional derivative $d\mathbf{f}/dx_i$) while the backward pass calculates for each sweep one row of J (gradient $\nabla_\mathbf{x} f_i$). Therefore, for $n < m$, forward mode needs fewer sweeps while reverse mode is preferable for $n > m$. Note that if f has a one dimensional output, reverse mode performs vector jacobian products (vjps) and thus calculates J in a single sweep. This is the case in machine learning, whenever learning success is quantified in a loss function.

C. Implementation in different languages

Two prominent ways, in which AD is implemented, are source to source transformation (see e.g. Python Jax, Julia Zygote, Fortran ADIFOR, C++ TAC++) and operator overloading (see e.g. Python Pytorch and tensor flow, C++ Adept). Source to source transformation starts from the source code that implements the forward pass and then generates and executes new source code that calculated derivatives. Packages that use operator overloading, store the functional

evaluation and the derivative for each performed operation. Mathematical operations and functions from the base language are overloaded for that task [1].

D. Dual numbers

The idea of storing the function value and its derivative can be motivated from a result of complex analysis (→ Cauchy-Riemann Equations) by which the derivative of a real valued function $g : \mathbb{R} \to \mathbb{R}$ is

$$g'(x) = \frac{Im(g(x+ih))}{h} + \mathcal{O}(h^2)$$

This equation can in fact be derived from the first order Taylor expansion $g(x+ih) = g(x) + g'(x)ih + \mathcal{O}(h^2)$ [2]. On the computer, the advantage over real number numerical differentiation $g'(x) \approx (g(x+h) - g(h))/h$ is that a) it is faster, because only one call of the function g is needed and b) it is more precise, because x and ih are stored separately on the machine and thus are not subject to roundoff error. To generalize this idea, a new type of number with two components (dual number $x+\epsilon$) is defined, and all operators are overloaded such that

$$g(x+\epsilon) = g(x) + \epsilon g'(x); \quad \epsilon^2 := 0 \wedge \epsilon \neq 0$$

For forward mode AD, this automatically implements the chain rule, because $g(x) + \epsilon g'(x)$ is again a dual number. For n dimensions, the number will have $n+1$ components and the coefficient of $\epsilon_i \epsilon_j$ will be the entry J_{ij}. Dual numbers are a central concept for implementing AD and will be used in future talks as well.

III. ADJOINT MODELS

First, let us clarify the domain specific terminology of "adjoints", and "adjoint models" [1][2]: The partial derivative of a function f^2 w.r.t its input f^1 is the same as the *adjoint* of the input w.r.t the function. As an example, the adjoint of f^1 is

$$f^{2\prime} = \bar{f^1} = \frac{\partial f^2}{\partial f^1} \tag{3}$$

For a model that calculates outputs from given inputs and parameters, one can construct an *adjoint model* that calculates the (total) derivatives of the output with respect to the inputs **efficiently**. The simplest case for an adjoint model is already shown above. For models that obtain the output by iterative transformation of the input, the adjoint model was substantially a product of jacobian matrixes multiplied in the most efficient order. However, if the model consists of an equation of some kind that needs to be solved, the accurate and computationally efficient adjoint model can generally also involve solving an equation. The following derivations for constructing adjoint models are inspired by the Chris Rackauckas online lecture [2].

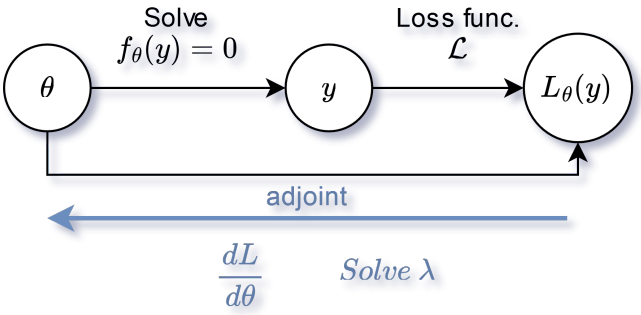

FIG. 1. Schematic diagram of nonlinear solver model. The adjoint model to get the derivative is a linear solver.

A. Nonlinear solve

Consider a parametrized model that consists of a nonlinear solve. The result is passed into a loss function.

$$y = \text{Solve}(f_\theta(y') = 0) \tag{4}$$
$$L = L_\theta(y) \tag{5}$$

Where $y \in \mathbb{R}^n$, $f_\theta : \mathbb{R}^n \to \mathbb{R}^n$ and $L_\theta : \mathbb{R}^n \to \mathbb{R}$. First, consider

$$\begin{aligned} f_\theta(y) &= 0 \\ \Rightarrow \frac{df}{d\theta} &= \frac{\partial f}{\partial y}\frac{\partial y}{\partial \theta} + \frac{\partial f}{\partial \theta} = 0 \\ \Rightarrow \frac{\partial y}{\partial \theta} &= -\left(\frac{\partial f}{\partial y}\right)^{-1}\frac{\partial f}{\partial \theta} \end{aligned}$$

The total derivative of the loss w.r.t. the parameters is given by

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial y}\frac{\partial y}{\partial \theta} \tag{6}$$
$$= \frac{\partial L}{\partial \theta} - \left(\frac{\partial L}{\partial y}\left(\frac{\partial f}{\partial y}\right)^{-1}\right)\frac{\partial f}{\partial \theta} \tag{7}$$

This solves the problem, mathematically, because all jacobians (written as partial derivatives) can be calculated from the model. However, to avoid the computationally heavy matrix inversion, one may define

$$\lambda^T := \frac{\partial L}{\partial y}\left(\frac{\partial f}{\partial y}\right)^{-1} \tag{8}$$

so that

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \lambda^T \frac{\partial f}{\partial \theta} \tag{9}$$

From its definition follows that λ can be obtained by solving the linear equation

$$\left(\frac{\partial f}{\partial y}\right)^T \lambda = \left(\frac{\partial L}{\partial y}\right)^T. \tag{10}$$

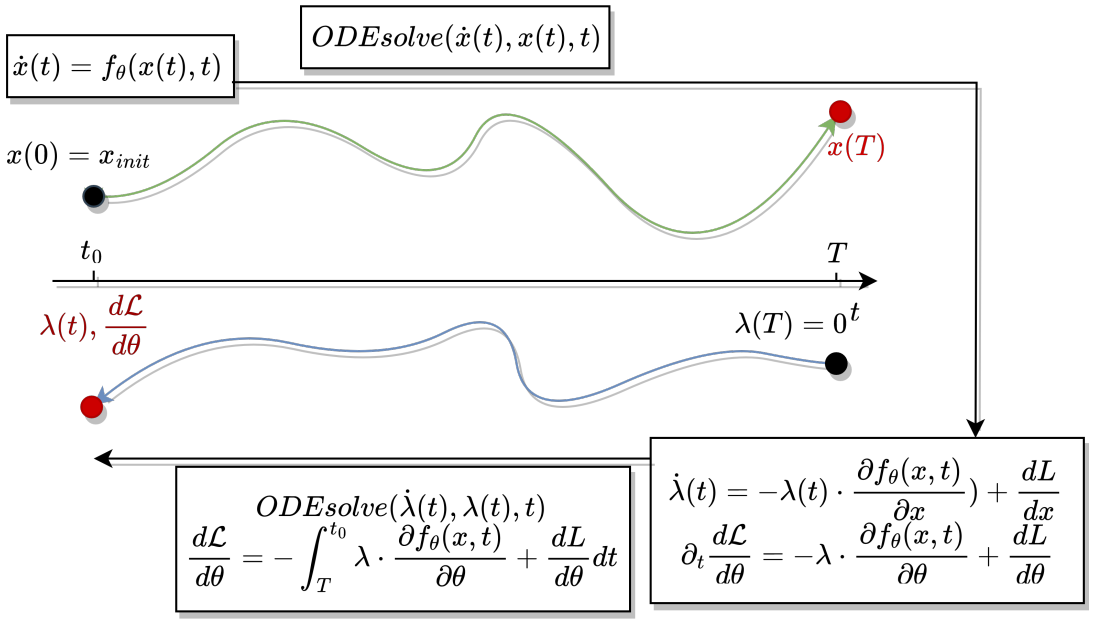

FIG. 2. Schematic process of the adjoint state method consisting of the forward pass (green), the backward pass (blue) and the corresponding equations in the boxes. For simplicity, this represents the 1D case with fixed initial conditions.

which in general is computationally more efficient than inverting the matrix. In summary, the adjoint model for Equation 4 consists of calculating the adjoints of θ and solving a linear equation involving the adjoints of y.

B. The Adjoint state method for ODEs

Now, the model is given by integrating a parametrized ODE,

$$\dot{x}(t) = f_\theta(x, t) \tag{11}$$

and the total loss function consists of a local loss function of the solution time series $x(t)$, integrated in time.

$$\mathcal{L}_\theta(x(t)) = \int_{t_0}^{T} L_\theta(x(t))dt$$

The goal now is to derive an adjoint model with which can be calculated the gradient $\frac{d\mathcal{L}_\theta}{d\theta}$. One can add a zero $0 = \dot{x}(t) - f_\theta(x, t)$ and multiply a time dependent vector $\lambda^T(t)$ in front. At this point, $\lambda^T(t)$ is free, and it can be restricted later on to yield the gradient.

$$\mathcal{L}(\theta) = \mathcal{L}_\theta - \int_{t_0}^{T} \lambda^T(t) \cdot (\dot{x}(t) - f_\theta(x(t), t))dt$$

Take the total derivative to get

$$\frac{d\mathcal{L}}{d\theta} = \int_{t_0}^{T} (\partial_\theta L + \partial_x L \partial_\theta x)dt - \int_{t_0}^{T} \lambda^T \cdot (\partial_\theta \dot{x} - \partial_x f \cdot \partial_\theta x - \partial_\theta f)dt$$

where time dependencies were dropped for simplicity. Performing partial integration on $\int_{t_0}^{T} \lambda^T \cdot \partial_\theta \dot{x}\ dt$ and rearranging terms yields

$$\begin{aligned}\frac{d\mathcal{L}}{d\theta} = &\int_{t_0}^{T} (\partial_\theta L + \lambda^T \partial_\theta f)dt \\ &+ |\lambda^T(t) \cdot \partial_\theta x(t)|_{t_0}^{T} \\ &- \int_{t_0}^{T} (\dot{\lambda}^T + \lambda^T \cdot \partial_x f - \partial_x L)\partial_\theta x dt\end{aligned}$$

In the first line is an integral expression where all the partial derivatives (jacobians) $\partial_\theta f(x(t))$ and $\partial_\theta L(x(t))$ can be derived from an ODE solve forward in time. The initially free variable $\lambda(t)$, however, can be chosen precisely so that the second and third row are equal to zero or known constants:

First, one can define an ODE for $\lambda(t)$, such that the third line evaluates to 0.

$$\dot{\lambda} = -\lambda \cdot \left(\frac{\partial f}{\partial x}\right)^T + \left(\frac{\partial L}{\partial x}\right)^T; \quad \lambda(T) = 0 \tag{12}$$

The freely chosen initial condition $\lambda(T) = 0$ simplifies the second line to $-\lambda^T(t_0)\partial_\theta x(t_0)$. Here, $\lambda(t_0)$ can be obtained by solving Equation 12 backwards in time. This leads to the result

$$\frac{d\mathcal{L}}{d\theta} = -\lambda^T(t_0)\partial_\theta x(t_0) + \int_{t_0}^{T} (\partial_\theta L + \lambda^T \cdot \partial_\theta f)dt \tag{13}$$

The total prcoess beginning with the initial conditions ending with the gradients is displayed in Figure 2. The

partial derivatives (jacobians) $\partial_\theta f(x(t))$ and $\partial_\theta L(x(t))$ are obtained by performing an ODE solve on Equation 11 forward in time with initial condition $x(t_0) = x_{input}$. Then, the ODE Equation 12 is solved backwards in time to get the *adjoint state* $\lambda(t)$. Finally, integrate the adjoint equation Equation 13.

Cost and accuracy: The backwards solve of the adjoint state allows making several tradeoffs: We can decide to save the complete time series $x(t)$ in the forward pass and reuse it, or only store $x(T)$ and solve the initial ODE, along with λ backwards in time. The first option saves memory while the second one prioritizes computational power, but can be instable for some ODE's. Secondly, in using ODE solvers we can select error margins, which trades off accuracy with computation time.

Assume that the ODE has $n = dim(f_\theta)$ dimensions and $p = dim(\theta)$ parameters. The computational efficiency can be compared to the straightforward finite difference approach, where $\dot{x} = f_\theta$ is solved for $\theta = (\theta_1, ..., \theta_i, ..., \theta_p)$ and $\theta' = (\theta_1, ..., \theta_i + h, ..., \theta_p)$ and the gradient $d\mathcal{L}/d\theta$ is given by the difference of the respective losses, divided by the small number h. For the total gradient, the finite difference approach needs to solve the n-dimensional ODE p times. By contrast, the adjoint state method only needs to solve one ODE forwards (n dimensions) and one ODE backwards (n dimensions for $\dot{\lambda}$ plus p dimensions for the integration).

Moreover, the aspect of precision also becomes central when dealing with exponential functions that often occur in ODE solutions: While the finite difference approach is severely limited by machine precision, the adjoint state method performs as precise as the ODE solver. This is also shown in our code example (https://github.com/Copani/ADjoint/tree/main).

C. Application of the adjoint state method

1. Neural ODE

As an application in machine learning, we can interpret the right-hand side function of Equation 11 as a Network. Since a Network is a nested function, it becomes an ODE with parameters that we can optimize for a desired output. As in Figure 3 visible, we have a normal ML problem with an input (initial value of time series $x(t_0)$, the black dot), a mapping (ODE forward solve, green line) and an output (final point of time series $x(T)$, red point). The output can be used to calculate a Loss and the gradient for optimization with Equation 13. This interpretation is useful if we deal with time series data. The ODE approach takes time correlation between data points into consideration by model choice. It performs especially well for sparse data or in interpolation tasks. One of the main drawbacks is the high computational cost [4].

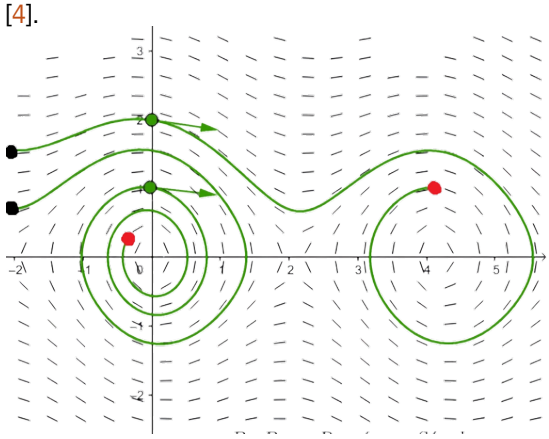

FIG. 3. ODE displayed as a phase space diagram - there is an input, initialising the ODE (black point), solve the ODE forward in time (green line) and output, as the point the ODE converged to (red point) | [3]

2. Seismology, higher derivatives and PDEs

The Adjoint state method is prominently used in seismic tomography, where refracted waves from earthquakes and eruptions are measured and compared to a model to reconstruct a 3D image of the subsurface of the earth. The two broad differences to the equations derived above are (1) that the state variables are distributed in space and thus described by a partial differential equation (PDE) and (2) that this PDE has the form of a wave equation which means that it is also of second order in time. It is possible to recover an ODE description by discretization in space and to reduce the order by adding dimensions $y = \dot{x}$. In so far, the adjoint state method for first order ODEs above is more general than it seems on first sight. In practice, however, it can be helpful to explicitly derive the adjoint state method for second order and express the discretized spacial operations by integrals (for further details see [5] and sources).

[1] C. C. Margossian, A review of automatic differentiation and its efficient implementation, WIREs Data Mining and Knowledge Discovery **9**, e1305 (2019).

[2] C. Rackauckas, Parallel computing and scientific machine learning (sciml): Methods and applications (2023).

[3] P. Rodríguez-Sánchez, Phase plane (2016).

[4] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, Neural ordinary differential equations, in *Advances in Neural Information Processing Systems*, Vol. 31, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018).

[5] A. M. Bradley, *PDE-constrained optimization and the adjoint method*, Tech. Rep. (Technical Report. Stanford University., 2013).