Simulation-Based Engineering Lab
University of Wisconsin-Madison
Technical Report TR–2023–08

Computing sensitivities of an Initial Value Problem via Automatic Differentiation: A Primer

Huzaifa Unjhawala, Ishaan Mahajan, Radu Serban, Dan Negrut

Department of Mechanical Engineering, University of Wisconsin – Madison

October 30, 2023

Contents

1 Introduction **2**

2 Sensitivity Analysis **2**

3 Continuous Sensitivity Analysis **3**
3.1 Forward Continuous Sensitivity Analysis 3
3.2 Adjoint Continuous Sensitivity Analysis 5

4 Discrete Sensitivity Analysis **7**

5 Discussion **10**

1 Introduction

Most dynamic systems, including vehicles can be modelled in the form of first order Initial Value problems which is an Ordinary Differential Equation (ODE) along with an Initial Condition. This in *explicit* form is usually given by:

$$\dot{\boldsymbol{u}} = f(\boldsymbol{u}, \boldsymbol{P}, t),\ \boldsymbol{u(t_0)} = \boldsymbol{u_0(P)} \tag{1}$$

where $\boldsymbol{u} \in \mathbb{R}^N$ where N is the dimension of the system state space and $\boldsymbol{P} \in \mathbb{R}^{N_P}$ is the vector of parameters that define the system. For a vehicle $\boldsymbol{P}$ consists of parameters such as tire stiffness, friction coefficient etc. It is not always the case that first/second order systems can be written in this form; but for the purpose of this report, we will only consider those systems that can.
In this technical report, we discuss the existing methods used for sensitivity analysis of first order systems given by Eq. (1) and where Automatic Differentiation (AD) can be used to make computation more accurate and fast. Section 2 gives a short overview on the taxonomy of different approaches uses for sensitivity calculation. The Continuous Sensitivity Analysis (CSA) is described in Sec. 3 along with a short discussion. Section 4 describes the Discrete Sensitivity Analysis along with its pros and cons in comparison to CSA. Discussion on the existing packages, available benchmarks and future work make up Sec. 5.

2 Sensitivity Analysis

Sensitivity analysis quantifies how the model output $\boldsymbol{u}$ changes for the change in model parameters $\boldsymbol{P}$. Such information is crucial for design optimization, parameter estimation, optimal control, data assimilation, process sensitivity, and experimental design [1]. These problems usually involve a cost functional ($J(\boldsymbol{P})$) that is minimized by varying model parameters. This cost functional can be expressed in two flavors:

1. in *continuous* form taking a continuous state trajectory $\boldsymbol{u(t)}$ as

$$J(\boldsymbol{P}) = \int_0^T g(\boldsymbol{u}, \boldsymbol{P}, t)\ dt \tag{2}$$

   or,

2. in *discrete* form taking a discrete sequence of the state trajectory $\boldsymbol{u} = [u_0, u_1, ...u_{N_T}]$ where N_T is the last time step.

$$J(\boldsymbol{P}) = \sum_{n=0}^{n=N_T} g_n(u_n, \boldsymbol{P}, n) \tag{3}$$

Most gradient based optimizers require the *total derivative* of the cost functional with respect to the parameters, $\frac{dJ}{d\boldsymbol{P}}$. The process of computing the total derivative using the continuous form Eq. (2) of the cost functional is termed *CSA* whereas if the discrete form Eq. (3) is used, the process is called *Discrete Sensitivity Analysis* (DSA) [2]. The difference in the two method is thus in the order in which we differentiate and discretize. In CSA, we first compute the sensitives and then we discretize to solve the ODE's whereas in DSA we compute the sensitives on the discretized equations. the Additionally, this total derivative can be computed in two different modes; (1) Forward mode, (2)

Adjoint mode. These define the taxonomy of the different approaches for sensitivity calculation and is shown in Fig. 1 [3].

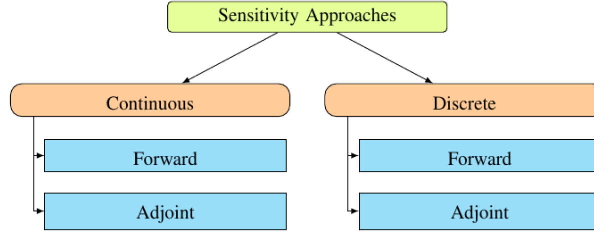

Figure 1: Shown is the taxonomy of approaches for sensitivity calculation. Note: the forward and adjoint approaches are in no way related to the forward and reverse modes in AD [3]

These different approaches are detailed in the upcoming sections of this technical report. It is important to note here that these *forward* and *adjoint* modes have nothing to do with the *forward* and *reverse* mode's of AD. The way in which AD enters the picture will also be discussed in the upcoming sections.
The need for an elaborate approach to compute the total derivative of the cost functional boils down to the requirement of taking total derivatives of $\boldsymbol{u}$ with respect to $\boldsymbol{P}$ without having an *explicit* relation between $\boldsymbol{u}$ and $\boldsymbol{P}$. This will become more clear in the following sections when these derivatives are derived analytically.

3 Continuous Sensitivity Analysis

Taking the total derivative of the continuous form of the cost functional given in Eq. (2) with respect to the parameter vector $\boldsymbol{P}$ yields (dropping arguments),

$$\frac{dJ}{d\boldsymbol{P}} = \int_0^T \frac{dg}{d\boldsymbol{P}} = \int_0^T \frac{\partial g}{\partial \boldsymbol{u}}\frac{d\boldsymbol{u}}{d\boldsymbol{P}} + \frac{\partial g}{\partial \boldsymbol{P}} \tag{4}$$

The difficult to compute quantity in Eq. 4 is the term $\frac{d\boldsymbol{u}}{d\boldsymbol{P}}$ as we do not have an *explicit* relation between $\boldsymbol{u}$ and $\boldsymbol{P}$ but rather only an expression for the time derivative of $\boldsymbol{u}$, $\dot{\boldsymbol{u}} = f(\boldsymbol{u}, \boldsymbol{P}, t)$, $\boldsymbol{u(t_0)} = \boldsymbol{u_0(P)}$. There are two approaches to deal with this quantity;

3.1 Forward Continuous Sensitivity Analysis

The easiest way to deal with $\frac{d\boldsymbol{u}}{d\boldsymbol{P}}$ is by applying the chain rule of differentiation to the original ODEs Eq. (1)

$$\frac{d}{d\boldsymbol{P}}\left(\frac{d\boldsymbol{u}}{dt}\right) = \frac{\partial f}{\partial \boldsymbol{u}}\frac{d\boldsymbol{u}}{d\boldsymbol{P}} + \frac{\partial f}{\partial \boldsymbol{P}} \tag{5}$$

$$\frac{d}{dt}\left(\frac{d\boldsymbol{u}}{d\boldsymbol{P}}\right) = \frac{\partial f}{\partial \boldsymbol{u}}\frac{d\boldsymbol{u}}{d\boldsymbol{P}} + \frac{\partial f}{\partial \boldsymbol{P}} \tag{6}$$

$$\dot{\boldsymbol{S}} = \frac{\partial f}{\partial u}\boldsymbol{S} + \frac{\partial f}{\partial \boldsymbol{P}} \tag{7}$$

where $S = \frac{d\boldsymbol{u}}{d\boldsymbol{P}}$. Eq. (7) is a system of N_p, N dimensional ODEs with initial condition $\boldsymbol{S}(t_0) = \frac{\partial u_0}{\partial \boldsymbol{P}}$. These are called the *forward sensitivity equations* and are solved alongside the original N dimensional ODE system rendering the total size of the system as $N(N_p + 1)$. Once we solve for $\boldsymbol{u}$ and $\boldsymbol{S}$, using any suitable integration scheme, we plug these back into Eq. (4) to solve N_p one dimensional (time) integration problems.

Computing Aspects

The forward sensitivity analysis is a $\mathcal{O}(NN_p)$ and scales linearly with the number of parameters N_p. On the bright side, Eq. (7) is a linear ODE and is thus simpler to solve than the potentially non-linear system ODE. Additionally, the linear coefficient $(\frac{\partial f}{\partial u})$ is the jacobian of the RHS of the original ODE and can thus be recycled from the original ODE solve step considering it employs a Newton iteration. The original ODE and the appended sensitivity ODEs can then be solved using the same integrator while ensuring judicious calculation of the RHS jacobian. The current state of art integrators for forward sensitivity analysis CVODES [1] enable these capabilities along with other optimizations.

Role of AD

At this point, we know exactly the quantities that need to be calculated in order to obtain the sensitivities of the cost functional with respect to the parameters:

1. The system ODE's given by Eq. (1) : These can be solved using any integrator. If an implicit integrator is used, then the RHS state jacobian $(\frac{\partial f}{\partial \boldsymbol{u}})$ needs to be computed.

2. The system of forward sensitivity ODE's given by Eq. (7) : This can be solved with the same integrator used for the system ODE's. It however requires the evaluation of two jacobians; the RHS state jacobian $(\frac{\partial f}{\partial \boldsymbol{u}})$ and the RHS parameter jacobian $(\frac{\partial f}{\partial \boldsymbol{P}})$. If an implicit integrator is used, $\frac{\partial f}{\partial \boldsymbol{u}}$ is already computed and can be reused.

3. The cost functional integral given by Eq. (4) : This requires the computation of the cost function state $(\frac{\partial g}{\partial \boldsymbol{u}})$ and parameter jacobian $(\frac{\partial g}{\partial \boldsymbol{P}})$.

AD can be used to compute these jacobians. There are two major ways of doing AD: (1) the forward mode, (2) the reverse mode. The reverse mode is usually used when $N_p >> N$ or when the jacobian matrix is long and short [4]. More intricate details of the two methods is out of the scope of this report and is left to [4, 5].

AD Implementation : A Note

Additionally, there are different implementations of both forward and reverse mode AD. These are again broadly classified into two groups: (1) Source transformation and (2) Operator overloading. Source transformation involves taking the source code of a computer program that performs a desired function and applying a preprocessor that uses differentiation rules, including elementary operators and the chain rule. This process generates new source code that can calculate derivatives. The original source code for evaluation and the transformed code for differentiation are then compiled and executed simultaneously. However, a major drawback of source transformation is its

inability to handle advanced programming statements, such as while loops, C++ templates, and other object-oriented features, due to its reliance on compile-time information only [4, 5]. Some of the packages that implement source transformation are *clad* [6], *Tapenade* [7] and the GPU code compatible *Enzyme* [8].
Operator overloading is an implementation technique used in AD, where a new class of objects is introduced. These objects comprise both the value of a variable on the expression graph and a differential component. It's important to note that not all variables on the expression graph will be part of this new class. However, the root variables, which necessitate sensitivities, and all intermediate variables that depend on these root variables, either directly or indirectly, will be included in this class [4, 5]. One of the major drawbacks of operator overloading is the memory usage, especially in reverse mode AD. This usually makes it slower than source transformation but is much easier to develop and maintain and is applicable for a wider range of C++ code. Some popular tools that perform operator overloading are *Adept* (current state of the art) [9], zCppAD [10] and *ADOL-C* [11].
Recently, the robotics community has been exploring the integration of source transformation and operator overloading techniques, leveraging packages like *CppADCodeGen* [12]. These approaches involve converting the dynamic expression graph into optimized C code, resulting in the elimination of the overhead associated with the expression graph. The converted C code is highly efficient, suitable for real-time applications, and can be executed on various platforms such as micro-controllers, multi-threaded environments, and even GPUs. One drawback is that special CppAD operators need to be used for conditionals through which gradients need to be traced. However, such an implementation is currently the fastest way of implementing AD [4, 5].

3.2 Adjoint Continuous Sensitivity Analysis

The adjoint continuous sensitivity analysis aims to eliminate the need of computing the system of state sensitives ($\frac{d\boldsymbol{u}}{d\boldsymbol{P}}$). This is done by solving the cost functional minimization problem with the system ODE's as equality constraints.

$$\min_{\boldsymbol{P}} J(\boldsymbol{u}, \boldsymbol{P}) \tag{8}$$

$$\text{s.t. } \dot{\boldsymbol{u}} = f(\boldsymbol{u}, \boldsymbol{P}, t),\ \boldsymbol{u(t_0)} = \boldsymbol{u_0(P)} \tag{9}$$

Using *Lagrange multipliers* $\boldsymbol{\lambda(t)}$, we form an augmented objective function

$$\mathcal{L}(\boldsymbol{u}, \boldsymbol{P}, \boldsymbol{\lambda}) = J(\boldsymbol{u}, \boldsymbol{P}) + \int_0^T \boldsymbol{\lambda}^T(t) \left(f - \frac{\partial \boldsymbol{u}}{\partial t} dt \right) \tag{10}$$

Now taking the derivative of this with respect to the parameter vector $\boldsymbol{P}$

$$\begin{aligned} \frac{d\mathcal{L}}{d\boldsymbol{P}} = \int_0^T \frac{\partial g}{\partial \boldsymbol{P}} + \frac{\partial g}{\partial \boldsymbol{u}} \frac{d\boldsymbol{u}}{d\boldsymbol{P}} \\ + \boldsymbol{\lambda}^T(t) \left(\frac{\partial f}{\partial \boldsymbol{P}} + \frac{\partial f}{\partial \boldsymbol{u}} \frac{d\boldsymbol{u}}{d\boldsymbol{P}} - \frac{d}{dt} \frac{d\boldsymbol{u}}{d\boldsymbol{P}} dt \right) \end{aligned} \tag{11}$$

Taking the total state jacobian ($\frac{d\boldsymbol{u}}{d\boldsymbol{P}}$) common;

$$\begin{aligned}\frac{d\mathcal{L}}{d\boldsymbol{P}} = \int_0^T \frac{\partial g}{\partial \boldsymbol{P}} + \boldsymbol{\lambda}^T(t)\frac{\partial f}{\partial \boldsymbol{P}} \\ + \left(\frac{\partial g}{\partial \boldsymbol{u}} + \boldsymbol{\lambda}^T(t)\frac{\partial f}{\partial \boldsymbol{u}} - \boldsymbol{\lambda}^T(t)\frac{d}{dt}\right)\frac{d\boldsymbol{u}}{d\boldsymbol{P}}dt\end{aligned} \tag{12}$$

Using integration by parts to better represent the coefficient of $\frac{d\boldsymbol{u}}{d\boldsymbol{P}}$;

$$\begin{aligned}\int_0^T \boldsymbol{\lambda}^T(t)\frac{d}{dt}\frac{d\boldsymbol{u}}{d\boldsymbol{P}}dt = \left[\boldsymbol{\lambda}^T(t)\frac{d\boldsymbol{u}}{d\boldsymbol{P}}\right]_0^T \\ + \int_0^T \left(\frac{d\boldsymbol{\lambda}}{dt}\right)^T \frac{d\boldsymbol{u}}{d\boldsymbol{P}}dt\end{aligned} \tag{13}$$

Applying the limits of integration;

$$\begin{aligned}= \boldsymbol{\lambda}^T(0)\frac{d\boldsymbol{u}}{d\boldsymbol{P}}(0) - \boldsymbol{\lambda}^T(T)\frac{d\boldsymbol{u}}{d\boldsymbol{P}}(T) \\ + \int_0^T \left(\frac{d\boldsymbol{\lambda}}{dt}\right)^T \frac{d\boldsymbol{u}}{d\boldsymbol{P}}dt\end{aligned} \tag{14}$$

Substituting back into Eq. (12);

$$\begin{aligned}\frac{d\mathcal{L}}{d\boldsymbol{P}} = \int_0^T \frac{\partial g}{\partial \boldsymbol{P}} + \boldsymbol{\lambda}^T(t)\frac{\partial f}{\partial \boldsymbol{P}} \\ + \left(\frac{\partial g}{\partial \boldsymbol{u}} + \boldsymbol{\lambda}^T(t)\frac{\partial f}{\partial \boldsymbol{u}} + \left(\frac{d\boldsymbol{\lambda}}{dt}\right)^T\right)\frac{d\boldsymbol{u}}{d\boldsymbol{P}}dt \\ + \boldsymbol{\lambda}^T(0)\frac{d\boldsymbol{u}}{d\boldsymbol{P}}(0) - \boldsymbol{\lambda}^T(T)\frac{d\boldsymbol{u}}{d\boldsymbol{P}}(T)\end{aligned} \tag{15}$$

Now, since we use Lagrange multipliers, we can set them to any value we like. The coefficients of the difficult to compute terms are shown in blue and green. We proceed by setting the blue terms to 0 to eliminate having to compute $\frac{d\boldsymbol{u}}{d\boldsymbol{P}}$ and $\frac{d\boldsymbol{u}}{d\boldsymbol{P}}(T)$. The green terms remain as $\frac{d\boldsymbol{u}}{d\boldsymbol{P}}(0)$ is easy to compute as we already have the initial conditions. This gives us our *adjoint sensitivity equations*:

$$\begin{aligned}\frac{\partial g}{\partial \boldsymbol{u}} + \boldsymbol{\lambda}^T(t)\frac{\partial f}{\partial \boldsymbol{u}} + \left(\frac{d\boldsymbol{\lambda}}{dt}\right)^T = \mathbf{0}^T \\ \boldsymbol{\lambda^T(T} = \mathbf{0}^T\end{aligned} \tag{16}$$

Transposing;

$$\begin{aligned}\frac{\partial g}{\partial \boldsymbol{u}}^T + \boldsymbol{\lambda}(t)\frac{\partial f}{\partial \boldsymbol{u}}^T + \left(\frac{d\boldsymbol{\lambda}}{dt}\right) = \mathbf{0} \\ \boldsymbol{\lambda(T)} = \mathbf{0}\end{aligned} \tag{17}$$

Using Eq. (17), we can solve for $\boldsymbol{\lambda}$ backwards in time (starting at $t = T$) after solving for u using Eq. (1) forward in time. Then, the solution of $\boldsymbol{\lambda}$ is used in Eq. (11) to solve N_p one dimensional time integration problem. There are however a lot of implementation details to be fleshed out since the adjoint system of equations is solved backwards (terminal value problem) and thus, based on stability requirements, might require different time stepping. This might require the generation of the forward solution at the required time steps during the backward solve, which can be expensive. The current state of the art is CVODES which implements a *check pointing* approach that balances between speed and memory requirements providing an efficient implementation [1].

Computing Aspects

The major reason why the adjoint method is attractive is that the problem size does not scale with the number of parameters N_p. This is straight forward to see from Eq. (17) where the adjoint sensitivity equations are just an additional N dimensional system of equations. Thus, irrespective of the number of parameters for which we desire sensitivities, we will always be solving a problem that scales $\mathcal{O}(N)$. However, the adjoint method is not always the most efficient, especially for problems with small N_p due to the limitations described above for the backward solve.

Role of AD

The interesting part is that the role of AD is exactly the same for both the forward sensitivity and adjoint sensitivity methods i.e. the same quantities $\frac{\partial f}{\partial \boldsymbol{u}}$, $\frac{\partial g}{\partial \boldsymbol{u}}$, $\frac{\partial f}{\partial \boldsymbol{P}}$, $\frac{\partial g}{\partial \boldsymbol{P}}$ and $\frac{\partial u_0}{\partial \boldsymbol{P}}$ need to be computed. These can be computed using AD.

4 Discrete Sensitivity Analysis

In the discrete case we have;

$$J(\boldsymbol{P}) = \sum_{n=0}^{n=N_T-1} g_n(\boldsymbol{u}_n, \boldsymbol{P}, n) \tag{18}$$

In the interest of space, we will be only deriving here the adjoint discrete sensitivity equations.
As with the continuous case, we formulate the problem as an optimization problem with equality constraints. Except, the equality constraints are discretized equations of the system ODE Eq. (1). Thus, in this case, we first discretize the system and then differentiate. The optimization problem is formulated as:

$$\min_{\boldsymbol{P}} J(\boldsymbol{u}, \boldsymbol{P}) \tag{19}$$

$$\text{s.t. } \boldsymbol{u}_{n+1} = \boldsymbol{u}_n + f_n(\boldsymbol{u}_n, \boldsymbol{P}, t_n), \boldsymbol{u}(\boldsymbol{P}, t_0) = \boldsymbol{u}_0(\boldsymbol{P}) \tag{20}$$

Here, we assume an explicit integrator for the ease of derivation, but any integrator can be used. Using Lagrange multipliers $\boldsymbol{\lambda}$,

$$J = \boldsymbol{\mu}^T \left(\boldsymbol{u}_0(\boldsymbol{P}) - \boldsymbol{u}(\boldsymbol{P}, \boldsymbol{t_0}) \right) + \sum_{n=0}^{N_T-1} g_n + \boldsymbol{\lambda}_n^T (\boldsymbol{u}_{n+1} - \boldsymbol{u}_n - \boldsymbol{f}_n) \tag{21}$$

Here, we use the Lagrange multiplier $\boldsymbol{\mu}$ to deal with the initial condition equality constraint. This is ignored in the derivation of the continuous sensitivity case as it leads to a relatively simple solution of setting $\boldsymbol{\mu} = \boldsymbol{\lambda}(0)$ to eliminate the need for computing initial condition sensitivities.

Taking the derivative with respect to the parameters;

$$\frac{dJ}{d\boldsymbol{P}} = \boldsymbol{\mu}^T \left(\frac{d\boldsymbol{u}_0}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}}{d\boldsymbol{P}} \right) + \sum_{n=0}^{n=N_T-1} \frac{\partial g_n}{\partial \boldsymbol{P}} + \frac{\partial g_n}{\partial \boldsymbol{u}_n} \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} \tag{22}$$

$$+ \lambda_n^T \left[\frac{d\boldsymbol{u}_{n+1}}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} - \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{P}} - \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{u}_n} \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} \right] \tag{23}$$

From the above equation we can see that this derivation is very specific to the integrator used. Thus for each integrator, the adjoint sensitivity equations will be different in the discrete case; this is not the case for the continuous adjoint sensitivity equations. Rearranging the terms we get:

$$= \boldsymbol{\mu}^T \left(\frac{d\boldsymbol{u}_0}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}}{d\boldsymbol{P}} \right) + \sum_{n=0}^{n=N_T-1} \frac{\partial g_n}{\partial \boldsymbol{P}} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{P}} + \sum_{n=0}^{n=N_T-1} \lambda_n^T \left[\frac{d\boldsymbol{u}_{n+1}}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} \right] + \sum_{n=0}^{n=N_T-1} \left[\frac{\partial g_n}{\partial \boldsymbol{u}_n} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{u}_n} \right] \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} \tag{24}$$

Taking out the initial condition;

$$= \boldsymbol{\mu}^T \left(\frac{d\boldsymbol{u}_0}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}}{d\boldsymbol{P}} \right) \tag{25}$$

$$+ \sum_{n=0}^{n=N_T-1} \frac{\partial g_n}{\partial \boldsymbol{P}} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{P}}$$

$$+ \sum_{n=0}^{n=N_T-1} \lambda_n^T \left[\frac{d\boldsymbol{u}_{n+1}}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} \right]$$

$$+ \left[\frac{\partial g_0}{\partial \boldsymbol{u}_0} - \boldsymbol{\lambda}_0^T \frac{\partial \boldsymbol{f}_0}{\partial \boldsymbol{u}_0} \right] + \sum_{n=1}^{n=N_T-1} \left[\frac{\partial g_n}{\partial \boldsymbol{u}_n} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{u}_n} \right] \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}}$$

Playing around with the summation of the term in blue, we get:

$$\sum_{n=0}^{n=N_T-1} \lambda_n^T \left[\frac{d\boldsymbol{u}_{n+1}}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} \right] = \tag{26}$$

$$\boldsymbol{\lambda}_{N_T-1}^T \frac{d\boldsymbol{u}_{N_T}}{d\boldsymbol{P}} - \boldsymbol{\lambda}_0^T \frac{d\boldsymbol{u}_0}{d\boldsymbol{P}} - \sum_{n=1}^{N_T-1} \left[\boldsymbol{\lambda}_n - \boldsymbol{\lambda}_{\boldsymbol{n-1}} \right]^T \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}} \tag{27}$$

Substituting this back in Eq. (25);

$$= \boldsymbol{\mu}^T \left(\frac{d\boldsymbol{u}_0}{d\boldsymbol{P}} - \frac{d\boldsymbol{u}}{d\boldsymbol{P}} \right) \tag{28}$$

$$+ \sum_{n=0}^{n=N_T-1} \frac{\partial g_n}{\partial \boldsymbol{P}} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{P}}$$

$$+ \boldsymbol{\lambda}_{N_T-1}^T \frac{d\boldsymbol{u}_{N_T}}{d\boldsymbol{P}} - \boldsymbol{\lambda}_0^T \frac{d\boldsymbol{u}_0}{d\boldsymbol{P}} - \sum_{n=1}^{N_T-1} \left[\boldsymbol{\lambda}_n^T - \boldsymbol{\lambda}_{\boldsymbol{n-1}}^T \right]^T \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}}$$

$$+ \left[\frac{\partial g_0}{\partial \boldsymbol{u}_0} - \boldsymbol{\lambda}_0^T \frac{\partial \boldsymbol{f}_0}{\partial \boldsymbol{u}_0} \right] + \sum_{n=1}^{n=N_T-1} \left[\frac{\partial g_n}{\partial \boldsymbol{u}_n} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{u}_n} \right] \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}}$$

Rearranging the terms so that we obtain the coefficient of $\frac{d\boldsymbol{u}_n}{d\boldsymbol{P}}$ and $tdydx\boldsymbol{u}_0\boldsymbol{P}$;

$$= \left[\boldsymbol{\mu}^T - \boldsymbol{\lambda}_0^T + \frac{\partial g_0}{\partial \boldsymbol{u}_0} - \boldsymbol{\lambda}_0^T \frac{\partial \boldsymbol{f}_0}{\partial \boldsymbol{u}_0} \right] \frac{\partial \boldsymbol{u}_0}{\partial \boldsymbol{P}} \tag{29}$$

$$- \boldsymbol{\mu}^T \frac{\partial \boldsymbol{u}(t_0)}{\partial \boldsymbol{P}} + \boldsymbol{\lambda}_{N_T-1}^T \frac{d\boldsymbol{u}_{N_T}}{d\boldsymbol{P}}$$

$$+ \sum_{n=1}^{n=N_T-1} \left[\frac{\partial g_n}{\partial \boldsymbol{u}_n} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{u}_n} \right]$$

$$+ \left[\frac{\partial g_n}{\partial \boldsymbol{u}_n} - \lambda_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{u}_n} + \boldsymbol{\lambda}_n^T - \boldsymbol{\lambda}_{\boldsymbol{n-1}}^T \right] \frac{d\boldsymbol{u}_n}{d\boldsymbol{P}}$$

Again, picking Lagrange multipliers to eliminate the need to compute state sensitives we get (setting blue terms to 0);

$$\boldsymbol{\mu} = \frac{\partial g_0}{\partial \boldsymbol{u}_0} + \boldsymbol{\lambda}_0 \frac{\partial f_0}{\partial \boldsymbol{u}\ _0} + \boldsymbol{\lambda}_0 \tag{30}$$

$$\boldsymbol{\lambda}_{N_T-1} = \mathbf{0} \tag{31}$$

$$\boldsymbol{\lambda}_n - \boldsymbol{\lambda}_{n+1} = -\frac{\partial g_{n+1}}{\partial \boldsymbol{u}_{n+1}} + \boldsymbol{\lambda}_{n+1} \frac{\partial \boldsymbol{f}_{n+1}}{\partial \boldsymbol{u}_{n+1}} \tag{32}$$

On solving the sequence $\boldsymbol{\lambda}_{(N_T-1):0}$ (backwards), after solving the forward system state sequence $\boldsymbol{u}_{0:(N_T-1)}$, we can plug them back into the Eq. (22) to get;

$$\frac{dJ}{d\boldsymbol{P}} = \left[\frac{\partial g_0}{\partial \boldsymbol{u}_0} + \boldsymbol{\lambda}_0 \frac{\partial f_0}{\partial \boldsymbol{u}\ _0} + \boldsymbol{\lambda}_0 \right] \frac{\partial \boldsymbol{u}(t_0)}{\partial \boldsymbol{P}} \tag{33}$$

$$+ \sum_{n=0}^{n=N_T-1} \frac{\partial g_n}{\partial \boldsymbol{P}} - \boldsymbol{\lambda}_n^T \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{P}} \tag{34}$$

Role of AD

In the discrete case, we differentiate the discretized equations directly. Thus we need to compute $\frac{\partial \boldsymbol{f}_{n+1}}{\partial \boldsymbol{u}_{n+1}}$, $\frac{\partial \boldsymbol{g}_{n+1}}{\partial \boldsymbol{u}_{n+1}}$, $\frac{\partial \boldsymbol{f}_{n+1}}{\partial \boldsymbol{P}}$ and $\frac{\partial \boldsymbol{g}_{n+1}}{\partial \boldsymbol{P}}$. Using this makes it hard to use external integrators as these might not be compatible to automatic differentiation libraries. Some packages that offer discrete sensitives in C++ are *PETSc TSAdjoint* [13], *PyTorch* integrators [14] and *Enzyme* [8].

5 Discussion

There are thus a lot of options for sensitivity analysis. One is whether to differentiate and then discretize (continuous) or discretize and then differentiate (discrete). In each of this, we can either choose forward sensitivities or use the adjoint sensitivities. To compute the requirement terms in these formulations, we can use AD, derive them analytically or use Finite Differences. If we choose to use AD, we can either choose operator overloading, source transformation or a combination of the two. Within each, we can again do either forward mode AD or reverse mode AD. There are no ground rules or empirical results showing one approach better than the other for general applications. In [15], the discrete and continuous adjoint methods are compared. A Julia based integrator is used. The required Jacobins are computed with combinations of : (1) AD or analytical or numerical (2) Forward or reverse mode AD and (3) Operator overloading or source transformation. The type of systems used in the analysis cover stiff and non-stiff ODEs, large systems and small systems, and include a PDE discretization with a dimension N for testing the scaling of the methodologies. The results show a strong performance advantage for AD based discrete sensitivity analysis for forward-mode sensitivity analysis on sufficiently small systems (approx. < 100 parameters + ODEs), and an advantage for continuous adjoint sensitivity analysis for sufficiently large systems. Significantly, tape-based (operator overloading) reverse-mode automatic differentiation, when implemented in its pure form, did not exhibit satisfactory performance or scalability in the benchmarks conducted. The reason that these implementations were primarily

optimized for machine learning models that heavily rely on large-scale linear algebra operations, such as matrix multiplications. These operations effectively reduce the tape's size relative to the computational workload involved. However, when dealing with differential equations, which often involve nonlinear functions and scalar operations, the ratio between the tape handling and the computational work diminishes. As a result, the performance of pure tape-based differentiation becomes less competitive compared to alternative methods of derivative calculations. For vehicle models, it is probably a good idea to experiment the different methods available and find the ones that are the most efficient for a particular application and this will make up future work.

References

[1] R. Serban and A. C. Hindmarsh, "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, 2005.

[2] M. Betancourt, C. C. Margossian, and V. Leos-Barajas, "The discrete adjoint method: Efficient derivatives for functions of discrete sequences," 2020.

[3] H. Zhang, S. Abhyankar, E. Constantinescu, and M. Anitescu, "Discrete adjoint sensitivity analysis of hybrid dynamical systems with switching," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 5, pp. 1247–1259, 2017.

[4] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *WIREs Data Mining and Knowledge Discovery*, vol. 9, no. 4, p. e1305, 2019.

[5] A. Griewank and A. Walther, *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, second ed., 2008.

[6] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, and V. Ilieva, "Clad – Automatic Differentiation Using Clang and LLVM," vol. 608, p. 012055, IOP Publishing, may 2015.

[7] L. Hascoet and V. Pascual, "The tapenade automatic differentiation tool: Principles, model, and specification," *ACM Trans. Math. Softw.*, vol. 39, may 2013.

[8] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert, "Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[9] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in c++," *ACM Trans. Math. Softw.*, vol. 40, jul 2014.

[10] B. Bell, "CppAD: a package for C++ algorithmic differentiation," 20200610.

[11] A. Griewank, D. Juedes, and J. Utke, "Algorithm 755: Adol-c: A package for the automatic differentiation of algorithms written in c/c++," *ACM Trans. Math. Softw.*, vol. 22, p. 131–167, jun 1996.

[12] J. R. Leal, "CppADCodeGen." `github.com/joaoleal/CppADCodeGen`, 2017.

[13] H. Zhang, E. M. Constantinescu, and B. F. Smith, "Petsc tsadjoint: A discrete adjoint ode solver for first-order and second-order sensitivity analysis," *SIAM Journal on Scientific Computing*, vol. 44, no. 1, pp. C1–C24, 2022.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[15] C. Rackauckas, Y. Ma, V. Dixit, X. Guo, M. Innes, J. Revels, J. Nyberg, and V. Ivaturi, "A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions," *CoRR*, vol. abs/1812.01892, 2018.