2021 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-2369-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/HPEC49654.2021.9622796

A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions

Yingbo Ma
Julia Computing
Cambridge, Massachusetts, USA
Email: yingbo.ma@juliacomputing.com

Vaibhav Dixit
Julia Computing
Pumas-AI
Bangalore, India
Email: vaibhav.dixit@juliacomputing.com

Michael J Innes
Edinburgh, UK
Email: mike.j.innes@gmail.com

Xingjian Guo
New York University
New York City, New York, USA
Email: xg703@nyu.edu

Chris Rackauckas
Massachusetts Institute of Technology
Julia Computing
Pumas-AI
Cambridge, Massachusetts, USA
Email: chris.rackauckas@juliacomputing.com

Abstract*—Derivatives of differential equation solutions are commonly for parameter estimation, fitting neural differential equations, and as model diagnostics. However, with a litany of choices and a Cartesian product of potential methods, it can be difficult for practitioners to understand which method is likely to be the most effective on their particular application. In this manuscript we investigate the performance characteristics of Discrete Local Sensitivity Analysis implemented via Automatic Differentiation (DSAAD) against continuous adjoint sensitivity analysis. Non-stiff and stiff biological and pharmacometric models, including a PDE discretization, are used to quantify the performance of sensitivity analysis methods. Our benchmarks show that on small stiff and non-stiff systems of ODEs (approximately < 100 parameters+ODEs), forward-mode DSAAD is more efficient than both reverse-mode and continuous forward/adjoint sensitivity analysis. The scalability of continuous adjoint methods is shown to be more efficient than discrete adjoints and forward methods after crossing this size range. These comparative studies demonstrate a trade-off between memory usage and performance in the continuous adjoint methods that should be considered when choosing the technique, while numerically unstable backsolve techniques from the machine learning literature are demonstrated as unsuitable for most scientific models. The performance of adjoint methods is shown to be heavily tied to the reverse-mode AD method used for the vector-Jacobian product calculations, with tape-based AD methods shown to be 2 orders of magnitude slower on nonlinear partial differential equations than static AD techniques. In addition, these results demonstrate the out-of-the-box applicability of DSAAD to differential-algebraic equations, delay differential equations, and hybrid differential equation systems where the event timing and effects are dependent on model parameters, showcasing an ease of implementation advantage for DSAAD approaches. Together, these benchmarks provide a guide to help practitioners to quickly identify the best mixture of continuous sensitivities and automatic differentiation for their applications.

I. Introduction

In the literature of differential equations, local sensitivity analysis is the practice of calculating derivatives to a differential equation's solution with respect to model parameters. For an ordinary differential equation (ODE) of the form

$$\dot{u} = f(u, p, t), \tag{1}$$

where f is the derivative function and p are the model parameters, the sensitivity of the state vector u with respect to model parameter p_i at time t is defined as $\frac{\partial u(p,t)}{\partial p_i}$. These sensitivities have many applications. For example, they can be directly utilized in fields such as biological modeling to identify parameters of interest for tuning and experimentation [1]. Recent studies have utilized these sensitivities as part for training neural networks associated with the ODEs [2], [3]. In addition, these sensitivities are indirectly utilized in many disciplines for parameter estimation of dynamical models. Parameter estimation is the problem of finding parameters p such that a cost function $C(p)$ is minimized (usually some fit against data) [4], [5], [6], [7], [1], [8], [9]. Gradient-based optimization methods require the computation of gradients of $C(p)$. By the chain rule, $\frac{dC}{dp}$ requires the calculation of $\frac{du(t_i)}{dp}$ which are the model sensitivities. Given the high computational cost of parameter estimation due to the number of repeated numerical solutions which are required, efficient and accurate computation of model sensitivities is an important part of differential equation solver software.

The simplest way to calculate model sensitivities is to utilize numerical differentiation which is given by the formula

$$\frac{\partial u(t)}{\partial p_i} = \frac{u(p + \Delta p_i, t) - u(p, t)}{\Delta p_i} + \mathcal{O}(\Delta p_i), \tag{2}$$

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

where $p + \Delta p_i$ means adding Δp_i to only the ith component of p. However, this method is not efficient (it requires two numerical ODE solutions for each parameter i) and it is prone to numerical error. If Δp_i is chosen too large, then the error term of the approximation is large. In contrast, if Δp_i is chosen too small, then calculations may exhibit floating point cancellation which increases the error [10].

To alleviate these issues, many differential equation solver softwares implement a form of sensitivity calculation called continuous local sensitivity analysis (CSA) [11], [12]. Forward-mode continuous sensitivity analysis calculates the model sensitivities by extending the ODE system to include the equations:

$$\frac{d}{dt}\left(\frac{\partial u}{\partial p_i}\right) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial p_i} + \frac{\partial f}{\partial p_i} \tag{3}$$

where $\frac{\partial f}{\partial u}$ is the Jacobian of the derivative function f with respect to the current state, and $\frac{\partial f}{\partial p_i}$ is the derivative of the derivative function with respect to the ith parameter. We note that $\frac{\partial f}{\partial u}v$ is equivalent to the directional derivative in the direction of v, which can thus be calculated Jacobian-free via:

$$\frac{\partial f}{\partial u}v \approx \frac{f(u + \epsilon v, p, t) - f(u, p, t)}{\epsilon} \tag{4}$$

or by equivalently pre-seeding forward-mode automatic differentiation with v in the dual space. Since the sensitivity equations for each i are dependent on the current state u, these ODEs must be solved simultaneously with the ODE system $u' = f$. By solving this expanded system, one can ensure that the sensitivities are computed to the same error tolerance as the original ODE terms, and only a single numerical ODE solver call is required.

However, since the number of ODEs in this system now scales proportionally with the number of parameters, the scaling of the aforementioned method is $\mathcal{O}(np)$ for n ODEs and p parameters, forward-mode CSA is not practical for a large number of parameters. Instead, for these cases continuous adjoint sensitivity analysis (CASA) is utilized. This methodology is defined to directly compute the gradient of a cost function of the solution with scaling $\mathcal{O}(n + p)$. Given a cost function c on the ODE solution which is evaluated as discrete time points (such as an L^2 loss)

$$C(u(p), p) = \sum_i c(u(p, t_i), p) \tag{5}$$

this is done by solving a backwards ODE known as adjoint problem

$$\frac{d\lambda'}{dt} = -\lambda'\frac{\partial f(u(t), p, t)}{\partial u} \tag{6}$$

where at every time point t_i, this backwards ODE is perturbed by $\frac{\partial c(u(p,t_i),p)}{\partial u}$. Note $u(t)$ is generated by a forward solution. The gradient of the cost function is then given by the integral:

$$\frac{dC}{dp} = \lambda'(t_0)\frac{\partial f(u(t_0), p, t_0)}{\partial u} + \sum_i \int_{t_i}^{t_{i+1}} \lambda'\frac{\partial f(u(t), p, t)}{\partial p} + \frac{\partial c(u(p, t_i), p)}{\partial p} dt \tag{7}$$

This integral may be solved via quadrature on a continuous solution of the adjoint equation or by appending the quadrature variables to perform the integration as part of the backsolve. The former can better utilize quadrature points to reduce the number of evaluation points required to reach a tolerance, but requires enough memory to store a continuous extension of the backpass. We note that the backsolve technique is numerically unstable, which will be noted in benchmarks where it produces a divergent calculation of the gradient. We note that, similarly to forward-mode, the crucial term $\lambda'(t)\frac{\partial f(u(t),p,t)}{\partial p}$ can be computed without building the full Jacobian. However, in this case this computation $v'J$ cannot be matrix-free computed via numerical or forward-mode differentiation, but instead is the primitive of reverse-mode automatic differentiation. Thus efficient methods for CASA necessarily mix a reverse-mode AD into the generated adjoint pass, and thus we will test the Cartesian product of the various choices.

In contrast to CSA methods, discrete sensitivity analysis calculates model sensitivities by directly differentiating the numerical method's steps [11]. However, this approach requires specialized implementations of the first order ODE solvers to propagate said derivatives. Instead, one can achieve the same end by using automatic differentiation (AD) on a solver implemented entirely in a language with pervasive AD, also known as a differentiable programming approach [13]. Section II introduces the discrete sensitivity analysis through AD (DSAAD) approach via type-specialization on a generic algorithm. Section III compares the performance of DSAAD against continuous sensitivity analysis and numerical differentiation approaches and shows that DSAAD consistently performs well on the tested models. Section IV describes limitations of the continuous sensitivity analysis approach and describes how these cases are automatically handled in the case of DSAAD. Together, this manuscript shows that the ability to utilize AD directly on numerical integrator can be advantageous to existing approaches for the calculation of model sensitivities, while at times can be disadvantageous in terms of scaling performance.

Currently, continuous sensitivity techniques are commonly used throughout software such as SUNDIALS [12], while some software like FATODE [11] allow for discrete sensitivity analysis. No previous manuscript performs comprehensive benchmarks on the Cartesian product of discrete/continuous forward/adjoint sensitivity analysis with the various automatic differentiation modes (tape vs static). This study uses DiffEqSensitivity.jl [14], the sensitivity analysis extension to DifferentialEquations.jl and the first comprehensive package which includes all mentioned differentiation choices, to do such a full comparison of the space. The results showcase the efficiency gains provided by discrete sensitivity analysis on

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

sufficiently small models, and establishes a heuristic range ($\approx$ 30-100 ODEs) at which the scalability of continuous adjoint sensitivity analysis overcomes the low overhead of DSAAD. This study can thus can be a central to helping all users of ODE solvers to choose the method that will be effective on their specific problem.

II. Discrete Sensitivity Analysis via Automatic Differentiation (DSAAD)

The core feature of the Julia programming language is multiple dispatch [15]. It allows a function to compile to different outputs dependent on the types of the inputs, effectively allowing choices of input types to trigger forms of code generation. ForwardDiff.jl provides a Dual number type which performs automatic differentiation on differentiable programs by simultaneously propagating a derivative along with the computed value on atomic (addition, multiplication, etc.) and standard mathematical (sin, exp, etc.) function calls [16]. By using the chain rule during the propagation, any function which is composed of differentiable calls is also differentiable by the methodology. Such a program is known as a differentiable program. Since this exactly differentiates the atomics, the numerical error associated with this method is similar to the standard evaluation of the function, effectively alleviating the errors seen in numerical differentiation. In addition, this method calculates derivatives simultaneously with the function's evaluation, making it a good candidate for fast discrete sensitivity analysis. It can be thought of as the analogue to continuous forward-mode sensitivity analysis on the set of differentiable programs. Similarly, reverse-mode AD from packages like ReverseDiff.jl [17] and Flux.jl [18] use Tracker numerical types which builds a tape of the operations and utilizes the reverse of the chain rule to "backpropogate" derivatives and directly calculate the gradient of some cost function. This implementation of AD can be thought of as the analogue of CASA on the set of differentiable programs.

The DifferentialEquations.jl package provides many integration routines which were developed in native Julia [19]. These methods are type-generic, meaning they utilize the numeric and array types that are supplied by the user. Thus these ODE solvers serve as a generic template whose internal operations can be modified by external packages via dispatch. When a generic DifferentialEquations.jl integrator is called with a Dual number type for the initial condition, the combination of these two programs results in a program which performs discrete sensitivity analysis. This combination is what we define as forward-mode DSAAD, and this combination with reverse-mode ReverseDiff.jl AD Tracker types is reverse-mode DSAAD.

To test the correctness of the DSAAD and continuous sensitivity methods, we check the outputted sensitivities on four models. For our tests we utilize nonlinear reaction models which are representative of those found in biological and pharmacological applications where these techniques are commonly used [20], [21]. The models are:

1) The non-stiff Lotka-Volerra equations (LV).

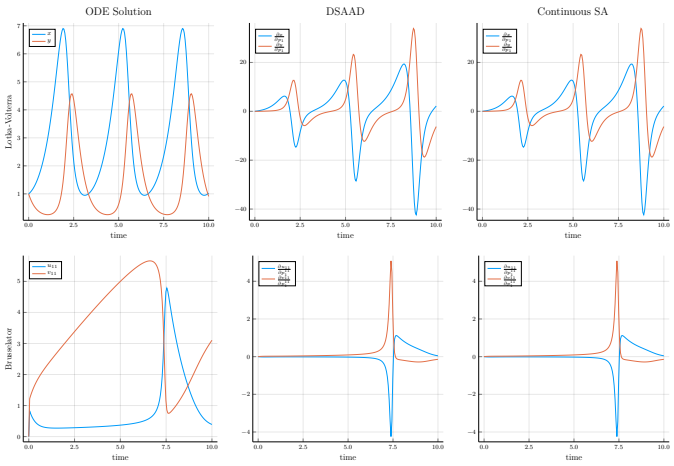

Fig. 1. **Model Sensitivities for DSAAD and CSA. Top row:** For the Lotka-Volterra model, the results are shown for $t \in [0, 10]$. **Bottom row:** The Brusselator PDE is discretized using a 3×3 uniform grid on the domain $[0, 1] \times [0, 1]$, and the resulting ODE system is solved for $t \in [0, 10]$. The results for u and v on the $(1, 1)$ grid point are shown.

2) An $N \times N$ finite difference discretization of the two-dimensional stiff Brusselator reaction-diffusion PDE (BRUSS).
3) A stiff pollution model (POLLU).
4) A non-stiff pharmacokinetic/pharmacodynamic system (PK/PD).

These 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. Details of the four models are presented in the Appendix.

Figure 1 shows the output of the first two models' sensitivities that are computed by the DSAAD method compared to CSA. The two methods align in their model sensitivity calculations, demonstrating that the application of AD on the generic ODE solver does produce correct output sensitivities. From these tests we note that the differences in the model sensitivities between the two methods had a maximum norm of 1.14×10^{-5} and 3.1×10^{-4} which is roughly the chosen tolerance of the numerical integration. These results were again confirmed at a difference of 1×10^{-12} when ran with sufficiently low ODE solver tolerance.

III. Benchmark Results

A. Forward-Mode Sensitivity Performance Comparisons

To test the relative performance of discrete and continuous sensitivity analysis, we utilized packages from the Julia programming language. The method for continuous sensitivity analysis is implemented in the DiffEqSensitivity.jl package by directly extending a user-given ordinary differential equation. This is done by extending the initial condition vector and defining a new derivative function $\tilde{f}$ which performs f on the first N components and adds the sensitivity equations. For performance, construction of the Jacobian can be avoided by utilizing AD for vector-Jacobian and Jacobian-vector products.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

Method/Runtime	LV (μs)	BRUSS (s)	POLLU (ms)	PKPD (ms)
DSAAD	174	1.94	12.2	2.66
CSA User-Jacobian	429	727	572	17.3
CSA AD-Jacobian	998	168	629	13.7
CSA AD-Jv seeding	881	189	508	8.46
Numerical Differentiation	807	1.58	24.9	17.4

TABLE I

FORWARD SENSITIVITY ANALYSIS PERFORMANCE BENCHMARKS. THE LOTKA-VOLTERRA MODEL USED BY THE BENCHMARKS IS THE SAME AS IN FIGURE 1, WHILE THE BRUSSELATOR BENCHMARKS USE A FINER 5×5 GRID WITH THE SAME SOLUTION DOMAIN, INITIAL VALUES AND PARAMETERS. THE TSIT5 INTEGRATOR IS USED FOR THE LOTKA-VOLTERRA AND THE PKPD MODEL. THE RODAS5 INTEGRATOR IS USED FOR THE BRUSSELATOR AND POLLU.

Method/Runtime	LV (μs)	BRUSS (s)	POLLU (s)	PKPD (ms)
Forward-Mode DSAAD	279	1.80	0.010	5.81
Reverse-Mode DSAAD	5670	19.1	0.194	133
CASA User-Jacobian (interpolating)	549	25.1	9.18	6.48
CASA AD-Jacobian (interpolating)	636	11.8	16.1	5.23
CASA AD-$v'J$ seeding (interpolating)	517	1.59	2.12	2.13
CASA User-Jacobian (quadrature)	693	0.964	1.82	4.88
CASA AD-Jacobian (quadrature)	825	2.17	6.19	4.97
CASA AD-$v'J$ seeding (quadrature)	707	0.461	1.30	2.94
CASA User-Jacobian (backsolve)	813	N/A	N/A	N/A
CASA AD-Jacobian (backsolve)	941	N/A	N/A	N/A
CASA AD-$v'J$ seeding (backsolve)	760	N/A	N/A	N/A
Numerical Differentiation	811	2.48	0.044	20.8

TABLE II

ADJOINT SENSITIVITY ANALYSIS PERFORMANCE BENCHMARKS. THE LOTKA-VOLTERRA AND BRUSSELATOR MODELS USED BY THE BENCHMARKS ARE THE SAME AS IN FIGURE 1. THE INTEGRATORS USED FOR THE BENCHMARKS ARE: RODAS5 FOR BRUSSELATOR AND POLLU, AND TSIT5 FOR LOTKA-VOLTERRA AND PKPD.

By seeding the Dual numbers to have partials v and applying forward applications of f, the resulting output is the desired $\frac{\partial f}{\partial u}v$. Similarly, seeding on Tracked reals for reverse-mode autodifferentiation results in $v'\frac{\partial f}{\partial u}$ which is the other desired quantity. As a comparison, full Jacobian implementations were also explored, either via a user-given analytical solution or automatic differentiation. We also include numerical differentiation performed by the FiniteDiff.jl package.

The test problems were solved with the various sensitivity analysis methods and the timings are given in Table I. In all of these benchmarks DSAAD performs well, being the fastest or nearly the fastest in all cases. While continuous sensitivity analysis does not necessarily require building the Jacobian, inspection of the generated LLVM from the forward mode reveals that the compiler is able to fuse more operations between the Jacobian-vector product and the other parts of the calculation, effectively improving common subexpression elimination (CSE) at the compiler level further than implementations which call functions for the separate parts of the calculation. Thus, given the equivalence of discrete forward sensitivities and forward-mode AD, these results showcase that sufficiently optimized forward-mode AD methods will be preferable in most cases.

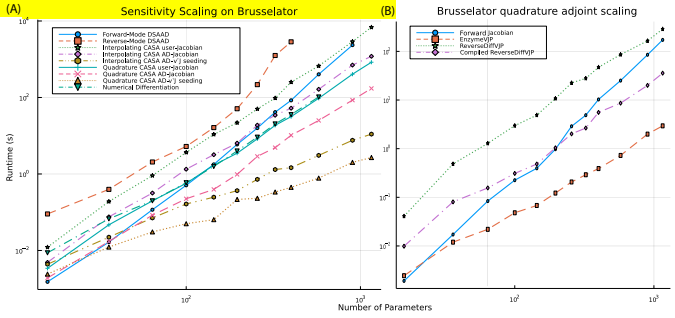

Fig. 2. **Brusselator Scaling Benchmarks.** A: The Brusselator problem was solved with varying dimension N with each of the different sensitivity analysis methods to determine their scalability with respect to number of parameters. Depicted on the x-axis is the number of parameters $4N^2$ in log-scale with the y-axis being the runtime in seconds in log-scale. All $v'J$ seeding uses Enzyme. B: The same Brusselator, comparing between reverse-mode AD techiques used for $v'J$ seeding.

B. Adjoint Sensitivity Performance Comparisons

For adjoint sensitivity analysis, adjoint DSAAD programs were produced using a combination of the generic DifferentialEquations.jl integrator with the tape-based automatic differentiation implementation of ReverseDiff.jl. DiffEqSensitivity.jl provides an implementation of CASA which saves a continuous solution for the forward pass of the solution and utilizes its interpolant in order to calculate the requisite Jacobian and gradients for the backwards pass. While this method is less memory efficient than checkpointing or re-solving schemes [12], it only requires a single forward numerical solution and thus was demonstrated as more runtime optimized for sufficiently small models. Thus while DifferentialEquations.jl contains a checkpointed adjoint implementation, checkpointing was turned off for the interpolating and backsolve schemes to allow them as much performance as possible.

The timing results of these methods on the test problems are given in Table II. These results show a clear performance advantage for forward-mode DSAAD over the other choices on sufficiently small models.

C. Adjoint Sensitivity Scaling

The previous tests all showed that on small models forward-mode via AD was advantageous to reverse-mode and adjoint methods. However, the advantage of adjoint methods comes in their ability to scale, with additive scaling with respect to the number of ODEs and parameters instead of multiplicative. Thus we decided to test the scaling of the methods on the Brusselator partial differential equation. For an $N \times N$ discretization in space, this problem has $2N^2$ ODE terms and $4N^2$ parameters. The timing results for the adjoint methods and forward-mode DSAAD are shown in Figure 2. Figure 2A demonstrates that as N increases, there is a point at which CASA becomes more efficient than DSAAD. This cutoff point seems to be around 50 ODEs+parameters when $v'J$ seeding is used and around 100 ODEs+parameters when $v'J$ seeding is not used with quadrature adjoints, to around 150 ODEs+parameters without $v'J$ seeding with interpolating adjoints (noted in the discussion section as the method equivalent to SUNDIALS [12]). This identifies a ballpark range of

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

around 100 combined ODEs+parameters at which practitioners should consider changing from forward sensitivity approaches to adjoint methods.

The most efficient CASA method is the quadrature-based method. This aligns with the theoretical understanding of the method. Stiff ODE solvers scale like $\mathcal{O}(n^3)$ where n is the number of ODEs. For the adjoint calculation, the quadrature based method has $n = 2N^2$, the number of ODEs of the forward pass. For the interpolating adjoint method, $n = 6N^2$, or the number of parameters plus the number of ODEs, which is then amplified by the cubic scaling of the linear solving. We note in passing that the backsolve technique requires $n = 8N^2$, or double the number of ODEs plus the number of parameters, but is unstable on stiff equations and thus is not effective on this type of problem (indeed, on these benchmarks it was unstable and thus omitted). We note that the inability for the reverse-mode DSAAD to scale comes from the tape generation performance on nonlinear models. The Tracker types in the differential equation solvers have to utilize scalar tracking instead of tracking array primitives, greatly increasing the size of the computational graph and the memory burden. When array primitives are used, mutation is not allowed which decreases the efficiency of the solver more than the gained efficiency. This is a general phenomena of tape-based automatic differentiation methods and has been similarly noted in PyTorch [22] and TensorFlow Eager [23], demonstrating that the effective handling of this approach would require alternative AD architectures. These issues may be addressed in the next generation reverse-mode source-to-source AD packages like Zygote [24] or Enzyme [25] by not relying on tape generation.

Figure 2B further highlights the importance of the reverse-mode architecture by comparing between different reverse-mode choices used for the $v'J$ calculation internal to the CASA. By default, ReverseDiff.jl is a tape-based AD which has trouble scaling to larger problems. Compiled ReverseDiff.jl does a single trace through the program to generate a static description of the code which it optimizes and compiles (this is thus not compatible with code that is non-static, such as having branches which change due to u or t). Enzyme.jl compiles a static SSA-form LLVM representation of the user's f function and uses compiler passes to generate the $v'J$ calculation. By acting at the lowest level, it can generate code after other code optimizations have been applied, which leads to noticeable performance gains of around an order of magnitude on this nonlinear partial differential equation application.

D. Parameter Estimation Performance Comparisons

To test the effect of sensitivity analysis timings in applications, we benchmarked parameter estimation performed with an L^2 loss function on generated data for each of the models. The data was generated by solving the ODE with the parameters defined in the Appendix and sampling at evenly spaced time points (100 points for Lotka-Volterra, 20 for Brusselator, 10 for POLLU, and 41 for PK/PD). Each of the parameter estimations were done using the BFGS local optimizer from Optim.jl [26] and were ran until the optimizer converged to the optima to a tolerance of 10^{-6} (results were checked for proper convergence). Each method started from the same initial condition which was a perturbation of the true parameters. For Lotka-Volerra, the initial parameter values were $\frac{4}{5}$ the true values, for PK/PD $0.95p_0 + 0.001$, and for the other models $\frac{9}{10}$. The timings are shown in Table III. While not as pronounced as the pure sensitivity calculations, these benchmarks show that utilizing a more efficient sensitivity analysis calculation does give a performance advantage in the application. Additional performance disadvantages for numerical differentiation could be attributed to the increased numerical error in the gradient which notably caused more iterations for the optimizer.

Method/Runtime	LV (s)	BRUSS (s)	POLLU (s)	PKPD (s)
Forward-Mode DSAAD	3.51	0.760	0.025	9.54
CSA User-Jacobian	1.47	52.2	0.931	12.3
CSA AD-Jacobian	1.82	51.9	0.932	12.5
CSA AD-Jv seeding	1.84	51.9	0.928	12.5
Reverse-Mode DSAAD	5.87	14.9	0.560	238
CASA User-Jacobian (interpolating)	0.030	12.4	0.238	42.5
CASA AD-Jacobian (interpolating)	0.031	5.58	0.414	25.0
CASA AD-$v'J$ seeding (interpolating)	0.028	2.26	0.060	13.4
CASA User-Jacobian (quadrature)	0.122	1.37	0.053	21.2
CASA AD-Jacobian (quadrature)	0.078	1.67	0.144	6.35
CASA AD-$v'J$ seeding (quadrature)	0.065	1.20	0.038	18.3
CASA User-Jacobian (backsolve)	2.95	N/A	N/A	N/A
CASA AD-Jacobian (backsolve)	1.35	N/A	N/A	N/A
CASA AD-$v'J$ seeding (backsolve)	1.30	N/A	N/A	N/A
Numerical Differentiation	0.105	8.17	0.110	168

TABLE III
PARAMETER ESTIMATION BENCHMARKS. THE LOTKA-VOLTERRA MODEL USED BY THE BENCHMARKS IS THE SAME AS IN FIGURE 1, WHILE THE BRUSSELATOR BENCHMARKS USE A FINER 5×5 GRID WITH THE SAME SOLUTION DOMAIN, INITIAL VALUES AND PARAMETERS. THE TSIT5 INTEGRATOR IS USED FOR LOTKA-VOLTERRA AND PKPD, AND THE RODAS5 INTEGRATOR IS USED FOR BRUSSELATOR AND POLLUTION.

IV. DSAAD GENERALIZES TO HYBRID, DELAY, AND DIFFERENTIAL-ALGEBRAIC DIFFERENTIAL EQUATIONS

We compared the flexibility of sensitivity analysis approaches in order to understand their relative merits for use in a general-purpose differential equation package. First we analyze the ability for the sensitivity analysis approaches to work with event handling. Event-handling is a feature of differential equation packages which allows users to provide a rootfinding function $g(u, p, t)$ at which a discontinuity (of the user's choice) is applied at every time point where $g(u, p, t) = 0$ (such equations are also known as hybrid differential equations). Automatic differentiation approaches to discrete sensitivity analysis directly generalize to handling this case by propagating Dual numbers through the event handling code. On the other hand, continuous sensitivity analysis approaches can require special handling in order to achieve correctness. There are two ways which sensitivities will not propagate:

1) Standard continuous sensitivity analysis does not take into account the sensitivity of the time point of the discontinuity to the parameters.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

Method	$\partial x(1)/\partial a$	$\partial y(1)/\partial a$	$\partial x(1)/\partial b$	$\partial y(1)/\partial b$
Analytical Solution	-1.0	-0.25	0	0.5
DSAAD	-1.0	-0.25	5.50e-11	0.5
CSA	-1.0	0.0	0.0	1.0

TABLE IV

Sensitivity Analysis with Events. SHOWN ARE THE RESULTS OF THE CONTROL PROBLEM GIVEN BY EQUATION 8 WITH $a = 2$ AND $b = 1$. IT WAS SOLVED ON $t \in [0, 1]$ WITH INITIAL CONDITION $(x(0), y(0)) = (1, 0)$. THE SENSITIVITIES OF THE TWO STATE VARIABLES ARE GIVEN AT TIME $t = 1$ WITH RESPECT TO THE TWO PARAMETERS. THE ANALYTICAL SOLUTION IS DERIVED BY TAKING DERIVATIVES DIRECTLY ON EQUATION 9.

2) Standard continuous sensitivity analysis does not take into account the possibility of the discontinuity's amount being parameter-dependent.

These points can be illustrated using a single state linear control problem where x is the signal responsible for the control of y. This results in a first-order linear hybrid ordinary differential equation system:

$$\begin{aligned} \frac{dx}{dt} &= -a, \\ \frac{dy}{dt} &= b, \end{aligned} \tag{8}$$

where $a, b > 0$ are parameters, and the rootfinding function is $g(x, y, p, t) = x$. At zero-crossings the parameter b is set to 0, effectively turning off the second equation. For the initial condition $(x(0), y(0)) = (1, 0)$, the system's analytical solution is

$$\begin{aligned} x(t) &= 1 - at, \\ y(t) &= \begin{cases} bt & (t < t^*) \\ bt^* = \frac{b}{a} & (t \geq t^*), \end{cases} \end{aligned} \tag{9}$$

where $t^* = 1/a$ is the crossing time, which depends on the parameters. Furthermore, the amount of jump for the discontinuity of dy/dt is also parameter dependent.

The sensitivity analysis results are compared to the true derivative at time $t = 1$ utilizing the analytical solution of the system (Equation 9) in Table IV. These results show that continuous sensitivity analysis as defined in Equation 3 does not properly propagate the sensitivities due to discontinuities and this results in incorrect derivative calculations for hybrid ODE systems. Extensions to continuous sensitivity analysis are required in order to correct these errors [27]. These have been implemented in DiffEqSensitivity.jl, though we remark from experience that such corrections are very non-trivial to implement and thus demonstrate an ease of implementation advantage for DSAAD.

Additionally, the continuous sensitivity analysis equations defined in Equation 3 only apply to ordinary differential equations. It has been shown that a different set of equations is required for delay differential equations (DDEs) [28]

$$\frac{d}{dt}\frac{\partial u(t)}{\partial p_i} = \frac{\partial G}{\partial u}\frac{\partial u}{\partial p_i}(t) + \frac{\partial G}{\partial \tilde{u}}\frac{\partial u}{\partial p_i}(t-\tau) + \frac{\partial G}{\partial p_i}(t), \tag{10}$$

where $\frac{du(t)}{dt} = G(u, \tilde{u}, p, t)$ is the DDE system with a single fixed time delay τ, and differential-algebraic equations (DAEs) [12]

$$\frac{\partial F}{\partial u}\frac{\partial u}{\partial p_i} + \frac{\partial F}{\partial \dot{u}}\frac{\partial \dot{u}}{\partial p_i} + \frac{\partial F}{\partial p_i} = 0, \tag{11}$$

where $F(\dot{u}, u, p, t) = 0$ is the DAE system. On the other hand, the discrete sensitivity analysis approach implemented via automatic differentiation is not specialized to ordinary differentiation equations since it automatically generates the sensitivity propagation at the compiler-level utilizing the atomic operations inside the numerical integration scheme. Thus these types of equations, and expanded forms such as DDEs with state-dependent delays or hybrid DAEs, are automatically supported by the connection between DifferentialEquations.jl and Julia-based automatic differentiation packages. Tests on the DDE and DAE solvers confirm this to be the case.

V. DISCUSSION

Performant and correct sensitivity analysis is crucial to many applications of differential equation models. Here we analyzed both the performance and generalizability of the full Cartesian product of approaches. Our results show a strong performance advantage for automatic differentiation based discrete sensitivity analysis for forward-mode sensitivity analysis on sufficiently small systems, and an advantage for continuous adjoint sensitivity analysis for sufficiently large systems. Notably, pure tape-based reverse-mode automatic differentiation did not perform or scale well in these benchmarks. The implementations have been generally optimized for the usage in machine learning models which make extensive use of large linear algebra like matrix multiplications, which decrease the size of the tape with respect to the amount of work performed. Since differential equations tend to be defined by nonlinear functions with scalar operations, the tape handling to work ratio thus decreases and is no longer competitive with other forms of derivative calculations. This technical detail affects common frameworks such as ReverseDiff.jl, PyTorch [22], and TensorFlow Eager [23], which leads to the recommendation of CASA for most frameworks. In addition, some frameworks, such as Jax [29], cannot JIT optimize the non-static computation graphs of a full ODE solver, which further leads to performance improvements of CASA in current reverse-mode implementations. Future work should investigate adjoint DSAAD with approaches which allow for low overhead compilation of the reverse path, such as that of Zygote [24], for which the static handling of control flow may be what is needed for this application.

One major result to note is that, in many cases of interest, runtime overhead of the adjoint methods can be larger than the theoretical scaling advantages. Forward-mode automatic differentiation does not exhibit the best scaling properties but on small systems of ODEs with small numbers of parameters, both stiff and non-stiff, the forward mode methods benchmarks as advantageous. On problems like PDEs, the scaling of adjoint methods does confer an advantage to them when the problem is sufficiently large but also a disadvantage when the problem

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

is small. Additionally, the ability to seed automatic differentiation for the Jacobian vector multiplications was demonstrated as very advantageous on the larger problems, showing that integrating the implementation of sensitivity analysis with automatic differentiation tools is necessary for achieving the utmost efficiency. In addition, the choice of automatic differentiation for the $v'J$ calculation was shown to be a major factor in performance, with the static Enzyme-based implementation giving a two order of magnitude performance advantage over the tape-based implementation even when confined to this one calculation. Together, these results show that having the ability to choose between these different methods is essential for a software wishing to support these distinct use cases.

Future research should look into the comparative performance of DiffEqSensitivity.jl [14] with the native adjoint techniques of SUNDIALS [12] and PETSc TS [30] to see how much this result generalizes. The results here would suggest the performance difference may be around three orders of magnitude since SUNDIALS uses a method similar to the interpolating CASA with (numerical) forward-mode Jacobians. A follow-up study on multiple PDEs comparing between the software suites could be very useful to practitioners.

While runtime performance of these methods is usually of interest, it is important to note the memory scaling for the various adjoint sensitivity analysis implementations. The quadrature CASA implementation benchmarks as the fastest for stiff ODEs, but uses a continuous solution to the original ODE in order to generate the adjoint Jacobian and gradients on-demand. This setup only requires a single forward ODE solve but makes a tradeoff due to the high memory requirement to save the full timeseries solution and its interpolating function. For example, with the chosen 9th order explicit Runge-Kutta method due to Verner, the total memory cost is $26NM$ since 26 internal derivative calculations of size N are utilized to construct the interpolant where M is the number of time points. The DSAAD reverse-mode AD approaches require constructing a tape for the entire initial forward solution and thus has memory scaling similar to the quadrature CASA, though in theory checkpointing AD systems AD could alleviate these memory issues. In contrast, checkpointing-based adjoint sensitivity analysis implementations [12] re-solve the forward ODE from a saved time point (a checkpoint) in order to get the u value required for the Jacobian and gradient calculation, increasing the runtime cost by at most 2x while decreasing the memory cost to NC where C is the number of checkpoints. These results show that the DiffEqSensitivity.jl interpolating checkpointing CASA approach may have around a 5x performance deficit over the more optimal quadrature CASA, which is can be a useful price to pay if memory concerns are the largest factor. The backsolve CASA approach, while the most memory efficient, was shown to be too unstable to be useful in most usecases and is thus only recommended if the dynamical system is known to be non-stiff in advance.

These results show many advantages for the AD-based discrete sensitivity analysis for small systems. However, there are significant engineering challenges to the development of such integration schemes. In order for this methodology to exist, a general function automatic differentiation tool must exist for the programming language and the entire ODE solver must be compatible with the AD software. This works for the Julia-based DifferentialEquations.jl software since it contains a large number of high performance native-Julia solver implementations. However, many other solver ecosystems do not have such possibilities. For example, common open source packages for solving ordinary differential equation systems include deSolve in R [31] and SciPy for Python [32]. While general automatic differentiation tools exist for these languages ([33], [34]), both of these package call out to Fortran-based ODE solvers such as LSODA [35], and thus AD cannot be directly applied to the solver calls. This means DSAAD techniques may stay limited to specific software due to technical engineering issues.

When considering the maintenance of large software ecosystems, the DSAAD approach gives many advantages if planned from the start. For one, almost no additional code was required to be written by the differential equation community in order for this implementation to exist since it directly works via code generation at compile-time on the generic functions of DifferentialEquations.jl. But an additional advantage is that this same technique applies to the native hybrid, delay, and differential-algebraic integrators present in the library. DifferentialEquations.jl also allows for many other actions to occur in the events. For example, the user can change the number of ODEs during an event, and events can change solver internals like the current integration time. Continuous sensitivity analysis require a separate implementation for each of these cases with could be costly to developer time. For DiffEqSensitivity.jl, the performant and correct implementation of corrections for hybrid differential equations and differential-algebraic equations took around 6 months of split between two developers (part time). This ease of implementation aspect may be worth considering for smaller organizations.

ACKNOWLEDGMENTS

This work is supported by Center for Translational Medicine, University of Maryland Baltimore School of Pharmacy. CR is partially supported by the NSF grant DMS1763272 and a grant from the Simons Foundation (594598, QN). Additionally, we thank the Google Summer of Code program for helping support the open source packages showcased in this manuscript. The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0001222 and NSF grants OAC-1835443 and IIP-1938400.

REFERENCES

[1] A. Sommer, "Numerical Methods for Parameter Estimation in Dynamical Systems with Noise," p. 282.

[2] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural Ordinary Differential Equations," *arXiv:1806.07366 [cs, stat]*, Jun. 2018, arXiv: 1806.07366. [Online]. Available: http://arxiv.org/abs/1806.07366

[3] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, "FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models," *arXiv:1810.01367 [cs, stat]*, Oct. 2018, arXiv: 1810.01367. [Online]. Available: http://arxiv.org/abs/1810.01367
[4] M. Peifer and J. Timmer, "Parameter estimation in ordinary differential equations for biochemical processes using the method of multiple shooting," *IET Systems Biology*, vol. 1, pp. 78–88, 2007.
[5] F. Hamilton, "Parameter Estimation in Differential Equations: A Numerical Study of Shooting Methods," *SIAM Undergraduate Research Online*, vol. 4, pp. 16–31, 2011. [Online]. Available: http://www.siam.org/students/siuro/vol4/S01073.pdf
[6] X. Zhen, "Parameter Estimation in Differential Equation Based Models," Ph.D. dissertation.
[7] C. Zimmer, "Parameter Estimation for Stochastic Models of Biochemical Reactions," *Journal of Computer Science & Systems Biology*, vol. 06, 2013.
[8] L. E. Friberg, A. Henningsson, H. Maas, L. Nguyen, and M. O. Karlsson, "Model of chemotherapy-induced myelosuppression with parameter consistency across drugs," *J Clin Oncol*, vol. 20, pp. 4713–21, Dec. 2002.
[9] B. Steiert, A. Raue, J. Timmer, and C. Kreutz, "Experimental design for parameter estimation of gene regulatory networks," *PLoS ONE*, vol. 7, p. e40052, 2012.
[10] R. L. Burden and J. D. Faires, *Numerical analysis*, 9th ed. Boston, MA: Brooks/Cole, Cengage Learning, 2011.
[11] H. Zhang and A. Sandu, "FATODE: A Library for Forward, Adjoint, and Tangent Linear Integration of ODEs," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C504–C523, Jan. 2014. [Online]. Available: http://epubs.siam.org/doi/10.1137/130912335
[12] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Trans. Math. Softw.*, vol. 31, pp. 363–396, 2005.
[13] C. Rackauckas, A. Edelman, K. Fischer, M. Innes, E. Saba, V. B. Shah, and W. Tebbutt, "Generalized physics-informed learning through language-wide differentiable programming." 2020.
[14] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman, "Universal differential equations for scientific machine learning," *arXiv preprint arXiv:2001.04385*, 2020.
[15] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, pp. 65–98, Jan. 2017.
[16] J. Revels, M. Lubin, and T. Papamarkou, "Forward-Mode Automatic Differentiation in Julia," *arXiv:1607.07892 [cs]*, Jul. 2016, arXiv: 1607.07892. [Online]. Available: http://arxiv.org/abs/1607.07892
[17] J. Revels, "ReverseDiff.jl." [Online]. Available: https://github.com/JuliaDiff/ReverseDiff.jl
[18] M. Innes, "Flux: Elegant Machine Learning with Julia," *Journal of Open Source Software*, 2018.
[19] C. Rackauckas and Q. Nie, "DifferentialEquations.jl - A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia," *Journal of Open Research Software*, vol. 5, p. 15, 2017.
[20] H. Kitano, "Computational systems biology."
[21] M. Danhof, E. C. M. de Lange, O. E. Della Pasqua, B. A. Ploeger, and R. A. Voskuyl, "Mechanism-based pharmacokinetic-pharmacodynamic (PK-PD) modeling in translational drug research," *Trends in Pharmacological Sciences*, vol. 29, no. 4, pp. 186–191, Apr. 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165614708000497
[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[23] A. Agrawal, A. N. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga *et al.*, "Tensorflow eager: A multi-stage, python-embedded dsl for machine learning," *arXiv preprint arXiv:1903.01855*, 2019.
[24] M. Innes, "Don't Unroll Adjoint: Differentiating SSA-Form Programs," *eprint arXiv:1810.07951*, p. arXiv:1810.07951, Oct. 2018.
[25] W. S. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients," in *Advances in Neural Information Processing Systems 33*, 2020, to appear in.
[26] P. K. Mogensen and A. N. Riseth, "Optim: A mathematical optimization package for Julia," *Journal of Open Source Software*, vol. 3, no. 24, p. 615, 2018.
[27] C. Kirches, "A Numerical Method for Nonlinear Robust Optimal Control with Implicit Discontinuities and an Application to Powertrain Oscillations," Ph.D. dissertation, Oct. 2006.
[28] H. T. Banks, D. Robbins, and K. L. Sutton, "Generalized Sensitivity Analysis for Delay Differential Equations," in *Control and Optimization with PDE Constraints*, K. Bredies, C. Clason, K. Kunisch, and G. von Winckel, Eds. Basel: Springer Basel, 2013, pp. 19–44.
[29] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax
[30] S. Abhyankar, J. Brown, E. M. Constantinescu, D. Ghosh, B. F. Smith, and H. Zhang, "Petsc/ts: A modern scalable ode/dae solver library," *arXiv preprint arXiv:1806.01437*, 2018.
[31] K. Soetaert, T. Petzoldt, and R. W. Setzer, "Solving Differential Equations in R: Package deSolve," *Journal of Statistical Software*, vol. 33, no. 9, 2010. [Online]. Available: http://www.jstatsoft.org/v33/i09/
[32] E. Jones, T. Oliphant, P. Peterson, and others, *SciPy: Open source scientific tools for Python*, 2001. [Online]. Available: http://www.scipy.org/
[33] S. E. Pav, "Madness: a package for Multivariate Automatic Differentiation," p. 15.
[34] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Autograd: Effortless Gradients in Numpy," p. 3.
[35] L. Petzold, "Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations."
[36] J. D. Murray, *Mathematical biology*, 3rd ed., ser. Interdisciplinary applied mathematics. New York: Springer, 2002.
[37] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II - Stiff and Differential-Algebraic Problems*. Springer, 1991.
[38] R. Upton and D. Mould, "Basic Concepts in Population Modeling, Simulation, and Model-Based Drug Development: Part 3: Introduction to Pharmacodynamic Modeling Methods," *CPT: Pharmacometrics & Systems Pharmacology*, vol. 3, no. 1, p. 88, Jan. 2014. [Online]. Available: https://doi.org/10.1038/psp.2013.71

VI. Appendix

A. Models

The first test problem is LV, the non-stiff Lotka-Volterra model

$$\begin{aligned}\frac{dx}{dt} &= p_1 x - p_2 xy,\\ \frac{dy}{dt} &= -p_3 y + xy.\end{aligned} \tag{12}$$

with initial condition $[1.0, 1.0]$ and $p = [1.5, 1.0, 3.0]$ [36]. The second model, BRUSS, is the two dimensional ($N \times N$) Brusselator stiff reaction-diffusion PDE:

$$\begin{aligned}\frac{\partial u}{\partial t} &= p_2 + u^2 v - (p_1 + 1)u + p_3(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) + f(x, y, t),\\ \frac{\partial v}{\partial t} &= p_1 u - u^2 v + p_4(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}),\end{aligned} \tag{13}$$

where

$$f(x, y, t) = \begin{cases} 5 & \text{if } (x - 0.3)^2 + (y - 0.6)^2 \leq 0.1^2 \text{ and } t \geq 1.1 \\ 0 & \text{else,} \end{cases} \tag{14}$$

with no-flux boundary conditions and $u(0, x, y) = 22(y(1 - y))^{3/2}$ with $v(0, x, y) = 27(x(1 - x))^{3/2}$ [37]. This PDE is discretized to a set of $N \times N \times 2$ ODEs using the finite difference method. The parameters are spatially-dependent, $p_i = p_i(x, y)$, making each discretized p_i a $N \times N$ set of values at each discretization point, giving a total of $4N^2$ parameters. The initial parameter values were the uniform $p_i(x, y) = [3.4, 1.0, 10.0, 10.0]$

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

The third model, POLLU, simulates air pollution. It is a stiff non-linear ODE system which consists 20 ODEs:

$$\begin{aligned}
\frac{du_1}{dt} &= -p_1u_1 - p_{10}u_{11}u_1 - p_{14}u_1u_6 - p_{23}u_1u_4 - \\
&\quad p_{24}u_{19}u_1 + p_2u_2u_4 + p_3u_5u_2 + p_9u_{11}u_2 + \\
&\quad p_{11}u_{13} + p_{12}u_{10}u_2 + p_{22}u_{19} + p_{25}u_{20} \\
\frac{du_2}{dt} &= -p_2u_2u_4 - p_3u_5u_2 - p_9u_{11}u_2 - p_{12}u_{10}u_2 + p_1u_1 + p_{21}u_{19} \\
\frac{du_3}{dt} &= -p_{15}u_3 + p_1u_1 + p_{17}u_4 + p_{19}u_{16} + p_{22}u_{19} \\
\frac{du_4}{dt} &= -p_2u_2u_4 - p_{16}u_4 - p_{17}u_4 - p_{23}u_1u_4 + p_{15}u_3 \\
\frac{du_5}{dt} &= -p_3u_5u_2 + p_4u_7 + p_4u_7 + p_6u_7u_6 + p_7u_9 + p_{13}u_{14} + p_{20}u_{17}u_6 \\
\frac{du_6}{dt} &= -p_6u_7u_6 - p_8u_9u_6 - p_{14}u_1u_6 - p_{20}u_{17}u_6 + p_3u_5u_2 + p_{18}u_{16} + p_{18}u_{16} \\
\frac{du_7}{dt} &= -p_4u_7 - p_5u_7 - p_6u_7u_6 + p_{13}u_{14} \\
\frac{du_8}{dt} &= p_4u_7 + p_5u_7 + p_6u_7u_6 + p_7u_9 \\
\frac{du_9}{dt} &= -p_7u_9 - p_8u_9u_6 \\
\frac{du_{10}}{dt} &= -p_{12}u_{10}u_2 + p_7u_9 + p_9u_{11}u_2 \\
\frac{du_{11}}{dt} &= -p_9u_{11}u_2 - p_{10}u_{11}u_1 + p_8u_9u_6 + p_{11}u_{13} \\
\frac{du_{12}}{dt} &= p_9u_{11}u_2 \\
\frac{du_{13}}{dt} &= -p_{11}u_{13} + p_{10}u_{11}u_1 \\
\frac{du_{14}}{dt} &= -p_{13}u_{14} + p_{12}u_{10}u_2 \\
\frac{du_{15}}{dt} &= p_{14}u_1u_6 \\
\frac{du_{16}}{dt} &= -p_{18}u_{16} - p_{19}u_{16} + p_{16}u_4 \\
\frac{du_{17}}{dt} &= -p_{20}u_{17}u_6 \\
\frac{du_{18}}{dt} &= p_{20}u_{17}u_6 \\
\frac{du_{19}}{dt} &= -p_{21}u_{19} - p_{22}u_{19} - p_{24}u_{19}u_1 + p_{23}u_1u_4 + p_{25}u_{20} \\
\frac{du_{20}}{dt} &= -p_{25}u_{20} + p_{24}u_{19}u_1
\end{aligned} \tag{15}$$

with the initial condition of $u_0 = [0, 0.2, 0, 0.04, 0, 0, 0.1, 0.3, 0.01, 0, 0, 0, 0, 0, 0, 0, 0.007, 0, 0, 0]^T$ and parameters $[.35, 26.6, 12, 300, .00086, .00082, 15,000, .00013, 24,000, 16,500, 9,000, .022, 12,000, 1.88, 16, 300, 4, 800,000, .00035, .0175, 10^9, .444e12, 1, 240, 2.1, 5.78, .0474, 1, 780, 3.12]$ [37].

The fourth model is a non-stiff pharmacokinetic/pharmacodynamic model (PKPD) [38], which is in the form of

$$\begin{aligned}
\frac{dDepot}{dt} &= -k_a Depot \\
\frac{dCent}{dt} &= k_a Depot + \\
&\quad (CL + V_{max}/(K_m + (Cent/V_c)) + Q_1)(Cent/V_c) + \\
&\quad Q_1(Periph_1/V_{p_1}) - \\
&\quad Q_2(Cent/V_c) + Q_2(Periph_2/V_{p_2}) \\
\frac{dPeriph_1}{dt} &= Q_1(Cent/V_c) - Q_1(Periph_1/V_{p_1}) \\
\frac{dPeriph_2}{dt} &= Q_2(Cent/V_c) - Q_2(Periph_2/V_{p_2}) \\
\frac{dResp}{dt} &= k_{in}(1 - (I_{max}(Cent/V_c)^\gamma/(IC_{50}^\gamma + (Cent/V_c)^\gamma))) - k_{out}Resp.
\end{aligned} \tag{16}$$

with the initial condition of $[100.0, 0.0, 0.0, 0.0, 5.0]$. $k_a = 1$ is the absorption rate of drug into the central compartment from the dosing compartment, $CL = 1$ is the clearance parameter of drug elimination, $V_c = 20$ is the central volume of distribution, $Q_1 = 2$ is the inter-compartmental clearance between central and first peripheral compartment, $Q_2 = 0.5$ is the inter-compartmental clearance between central and second peripheral compartment, $V_{p_1} = 10$ is the first peripheral compartment distribution volume, $V_{p_2} = 100$ is the second peripheral compartment distribution volume, $V_{max} = 0$ is the maximal rate of saturable elimination of drug, $K_m = 2$ is the Michaelis-Mentens constant, $k_{in} = 10$ is the input rate to the response (PD) compartment with a maximal inhibitory effect of $I_{max} = 1$, $IC_{50} = 2$ is a parameter for the concentration at 50% of the effect and $k_{out} = 2$ is the elimination rate of the response, and $\gamma = 1$ is the model sigmoidicity. Additional doses of 100.0 are applied to the $Depot$ variable at every 24 time units.

B. Hardware and Software Information

All the benchmark was run on Julia version 1.6.1 with the default OpenBLAS and on the machine with Intel(R) Xeon(R) Silver 4114 CPU with 64 GB of RAM using the Linux operating system.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE