Author version

Published as:

Sperry, M. Z., & Hwang, J. T. (2025). Ozone: An open-source ordinary differential equation solver for gradient-based optimization. Optimization and Engineering, 26(3), 2145-2183. https://doi.org/10.1007/s11081-025-09967-y

BibTeX for this reference:

https://lsdolab.github.io/lsdo-publications/publication/sperry2025ozone/

```
@article{sperry2025ozone,
  author = {Mark Z. Sperry and John T. Hwang},
  title = {Ozone: an open-source ordinary differential equation solver for
    gradient-based optimization},
  journal = {Optimization and Engineering},
  year = {2025},
  volume = {26},
  number = {3},
  pages = {2145-2183},
  doi = {10.1007/s11081-025-09967-y},
  url = {https://doi.org/10.1007/s11081-025-09967-y}
}
```

Noname manuscript No.
(will be inserted by the editor)

`Ozone`: an open-source ordinary differential equation solver for gradient-based optimization

Mark Z. Sperry[1] · **John T. Hwang**[1]

Received: date / Accepted: date

Abstract We present `Ozone`, a Python library for solving ordinary differential equations (ODEs) within gradient-based optimization algorithms. `Ozone` makes available the entire family of explicit and implicit Runge–Kutta methods and implements several solution approaches including parallel-in-time approaches. A unique feature is the ability to perform sensitivity analysis for any combination of method and solution approaches. `Ozone` is implemented as a library in the Computational System Design Language (`CSDL`), enabling the automatic computation of the derivatives of the objective and constraints across the numerical integration process . This paper describes the components of the software implementation of `Ozone` and demonstrates its features through four illustrative applications. `Ozone` allows researchers to easily implement and solve ODEs for optimal control and gradient-based multidisciplinary optimization.

Keywords Differential equations · Adjoint-based optimization · Parallel computing · Numerical methods

1 Introduction

Ordinary differential equations (ODEs) often appear in mathematical models of complex engineered systems, describing the evolution of the state of the system over time. The need to optimize parameters of models containing ODEs arises frequently during the analysis and design of such systems. Two classes of problems in this context are optimal control and multidisciplinary design optimization (MDO). Optimal control involves finding an ideal control input profile for an ODE to minimize or maximize a performance criterion under constraints. For example, minimum-time low-thrust orbit transfers can be obtained through optimal control to find the corresponding thrust profile (Graham and Rao, 2015). Another common application setting is the field of robotics, where the goal is often to determine optimal actuation profiles for the robot (e.g., Yan et al. (2019)). Optimal control is also used in chemical engineering such as for temperature control (Nolasco et al., 2021). On the other hand, MDO problems involve multiple discipline models and seek to optimize the design rather than control profiles. Discipline models in these problems often contain ODEs. For instance, MDO is applied in aircraft, wind turbine, and satellite design, where ODEs are used to model aircraft weight over missions, simulate wind turbine performance, and calculate satellite trajectories (Hwang et al., 2019; Deshmukh and Allison, 2016; Hwang et al., 2014).

Both optimal control and MDO problems are often solved using gradient-based optimization algorithms. These algorithms require not only the evaluation of objectives and constraints but also their derivatives in the optimization loop.

Mark Z. Sperry
E-mail: msperry@ucsd.edu

[1] Department of Mechanical and Aerospace Engineering, University of California San Diego, La Jolla, CA 92093, USA

However, computing derivatives of models that contain ODEs is non-trivial. Although methods for numerical integration are well-established, time and memory costs can be high if there are a large number of time steps or states. For derivative computation, the discrete adjoint method and tangent-linear methods are two efficient general-purpose techniques (Gray et al., 2019). However, applying them to a numerical integration procedure can be computationally expensive due to the need to solve potentially large linear systems and store state values at multiple or all time instances. Additionally, requiring partial derivatives of the ODE function can be a major obstacle if they are difficult to derive analytically. Finally, interfacing the large implementation of an ODE and their derivatives with other analysis models in the optimization problem can be a challenge if there are many subsystems.

Despite the aforementioned challenges, there are a number of widely used ODE solver software implementations with sensitivity analysis capabilities. DifferentialEquations.jl (Rackauckas and Nie, 2017) is a robust, widely used ODE solver implemented in Julia with a large selection of available methods. Another solver with built-in sensitivity analysis is FATODE (Zhang and Sandu, 2014), written in Fortran, which allows the use of Runge–Kutta and Rosenbrock methods. ODESSA computes sensitivities in conjuction with integration in Fortran. CVODES (Serban and Hindmarsh, 2003), part of the SUNDIALS suite, employs the backward differentiation formula or Adams–Moulton methods with checkpointing. Similarly, the DENSERKS library (Alexe and Sandu, 2009) also uses the checkpointing technique. TSAdjoint (Zhang et al., 2019) allows computation of first and second derivatives using checkpointing. Additionally, there are a number of popular solvers focusing on optimal control that employ parallel-in-time methods. GPOPS-II (Patterson and Rao, 2014) uses adaptive mesh refinement to increase efficiency. OpenMDAO's Dymos (Falck et al., 2021) allows analytic derivatives for sensitivity analysis. There are also specialized solvers such as SODECL (Avramidis et al., 2020) that is built for solving stochastic differential equations in a parallel environment. SciPy, a popular open-source mathematical toolkit for Python, offers a robust ODE solver with several explicit and implicit Runge-Kutta methods albeit without native support for automatic sensitivity analysis with respect to parameters (Virtanen et al., 2020). Finally, numerical integration routines with parameter sensitivities are available for several Python machine learning libraries such as Tensorflow's ODE module (Abadi et al., 2015) and torchdiffeq for Pytorch (Chen et al., 2018), although they are primarily designed for neural ODEs. However, to the best of our knowledge, there is currently no known software that allows accessible utilization of both parallel and serial integration techniques, while also providing automatic sensitivity analysis capabilities.

This paper presents `Ozone`, a Python library available at `https://github.com/LSDOlab/ozone` with the ability to integrate ODEs using a variety of methods and approaches for large-scale multidisciplinary optimization. `Ozone` makes available any method from the explicit or implicit Runge–Kutta family. These can be solved using one of four solution approaches: time-marching, time-marching with checkpointing, Picard iteration and collocation. Time-marching is the standard sequential integrator that solves for the state history by stepping through time. Time-marching with checkpointing uses the same technique, but significantly reduces memory usage at the cost of an increase in computation time. Picard iteration is a parallel-in-time approach that solves the ODE using fixed-point iteration. Collocation is another parallel-in-time approach but it formulates and solves an optimization problem to compute the states. `Ozone` provides a flexible interface that allows users to seamlessly switch out different numerical methods and solution approaches without the need to modify their code implementations.

A key feature of `Ozone` is the capability to automatically perform adjoint-based sensitivity analysis for any combination of methods and approaches. To achieve this unique capability, we implement the aforementioned methods and approaches and their derivatives using a general formulation referred to as the general linear methods (Butcher, 2016). `Ozone` is designed for use with the Computational System Design Language (`CSDL`), a novel algebraic modeling language (Gandarillas et al., 2024). `CSDL` automates the calculation of both the ODE function derivatives as well as the objective gradient and constraint Jacobian of the larger optimization model. Furthermore, `CSDL` provides an accessible interface to connect variables to other analysis models and the ability to execute models using different compiler backends, increasing computational performance compared to native Python. The aforementioned features are demonstrated through four applications.

We note that a prior version of Ozone was implemented by Hwang and Munster (2018). However, the implementation of Ozone presented here is a significant departure from the prior work due to the utilization of CSDL, a completely different system modeling framework. Unlike CSDL, OpenMDAO does not offer fully automatic derivative evaluation and model acceleration using different computational backends.

The new implementation also offers the time-marching with checkpointing approach, not offered by the previous version, which enables the solution of larger-scale problems. Finally, this implementation introduces a reformulation and reimplementation of all solution approaches and a redesigned API.

Our version of Ozone has been demonstrated in several recent studies. Orndorff et al. used Ozone for trajectory optimization to compute minimum-time and minimum-energy transfers from hover to forward flight for an electric air taxi (Orndorff and Hwang, 2022; Orndorff et al., 2023). Yan et al. used Ozone in a dynamic topology optimization algorithm for battery design (Yan et al., 2021). Further, Ozone successfully solves the lunar vehicle ascent problem defined by Sandoval et al (2024).

In the following section, we review the general linear methods, which is used as the mathematical foundation for the implementation of Ozone's integration methods as well as the adjoint method, which is used for the implementation of sensitivity analysis. Then, we describe the methods and algorithms used in Ozone in detail. Finally, we give a brief summary of the user interface, and go over four application problems for demonstration and benchmarking.

2 Background

Prior to presenting our methodology, we provide a brief overview of methods for numerically solving ordinary differential equations as well as methods for computing the derivatives of complex numerical models.

2.1 Numerical Methods for Ordinary Differential Equations

An ordinary differential equation (ODE) can be written as

$$\dot{\mathbf{y}} = f(\mathbf{y}, \mathbf{p}_d, \mathbf{p}_s), \qquad \mathbf{y}(t_0) = \mathbf{y}_0, \tag{1}$$

where $\mathbf{y} \in \mathbb{R}^{n_y}$ is the vector of states, $\mathbf{y}_0 \in \mathbb{R}^{n_y}$ is the initial condition, $\mathbf{p}_d \in \mathbb{R}^{n_{pd}}$ is the vector of dynamic parameters and $\mathbf{p}_s \in \mathbb{R}^{n_{ps}}$ is the vector of static parameters. States are quantities whose time histories are being solved for, and parameters are additional arguments to f that are not states. Static parameters are constant in time while dynamic parameters vary in time.

Other, more general types of differential equations can sometimes be represented as ODEs in the form of (1) such as partial differential equations (PDEs). The method of lines is one way to numerically reformulate a PDE into an ODE by discretizing a subset of variables (Sadiku and Obiozor, 2000). Differential algebraic equations (DAEs) defined by the form $\tilde{f}(\dot{\mathbf{y}}, \mathbf{y}, \mathbf{p}_d, \mathbf{p}_s) = 0$ can be solved using numerical methods for ODEs in some cases (Petzold, 1982). Stochastic differential equations (SDEs) are distinct from ODEs in that f is dependent on random processes. Their solutions can be represented as solutions of ODEs for SDEs in specific forms (Eugene and Moshe, 1965).

We can solve the initial value problem defined by (1) using a wide variety of numerical methods. Runge–Kutta methods are multi-stage methods, and they are given by

$$\mathbf{Y}_i = \mathbf{y}^{[n-1]} + h_n \sum_{j=1}^{s} a_{i,j} f(T_j, \mathbf{Y}_j), \qquad T_i = t_{n-1} + c_i h_n, \qquad i = 1, ..., s, \tag{2}$$

and

$$\mathbf{y}^{[n]} = \mathbf{y}^{[n-1]} + h_n \sum_{j=1}^{s} b_j f(T_j, \mathbf{Y}_j). \tag{3}$$

Runge–Kutta methods approximate y at the nth discretized point in time from the previous time index $n-1$ using s stages with step size h_n. The coefficients $a_{i,j}$, b_j, and c_j are elements of $A \in \mathbb{R}^{s\times s}$, $B \in \mathbb{R}^s$, and $C \in \mathbb{R}^s$, respectively, from the Butcher tableau (Butcher, 2016). The Butcher tableau uniquely defines a Runge–Kutta method and can be used to determine properties such as stability, and whether the method can be solved explicitly or implicitly. To integrate the ODE f to a final time $t_f = t_0 + \sum_{n=1}^{n_t} h_n$, Equation (3) is successively evaluated n_t times.

Another common method is the multi-step family defined by the general form

$$\begin{aligned}\mathbf{y}^{[n]} = &\gamma_1 \mathbf{y}^{[n-1]} + \ldots + \gamma_q \mathbf{y}^{[n-q]} \\ &+ h_n \beta_0 f(t_n, \mathbf{y}^{[n]}) + \ldots \\ &+ h_n \beta_q f(t_{n-q}, \mathbf{y}^{[n-q]}),\end{aligned} \tag{4}$$

which approximates $\mathbf{y}^{[n]}$ from the q solution states preceding n. β_i, γ_i are constant coefficients to polynomials that characterize a method (Butcher, 2016). Common multi-step methods include Adams–Bashforth methods, which are explicit, and Adams–Moulton methods, which are implicit. Finally, the backward differentiation formulas are also multi-step methods used to solve stiff differential equations.

Runge–Kutta and multi-step methods are similar in that they are generally solved sequentially: the solution is built up by marching through the time domain from the initial condition. Because of this, neither method is formulated to naturally take advantage of parallel computing.

2.2 General Linear Methods

Another major classification of methods is the general linear methods (GLM) family, originally presented by Butcher (Butcher, 2006). The GLM family includes all Runge–Kutta methods and all linear multi-step methods. Methods from the GLM family can be expressed as a single, unified form, making them convenient to adopt for general-purpose ODE solver libraries (Butcher, 2016). We do so in `Ozone`, where the benefits of a common form are even more important because of the need to compute derivatives across the entire time-integration process. The GLM matrix equation to solve for y at time steps n as a function of y at time steps $n-1$ can be written as

$$\begin{bmatrix}\mathbf{Y} \\ \mathbf{y}^{[n]}\end{bmatrix} = \begin{bmatrix}\mathbf{A}\otimes\mathcal{I}_{n_y} & \mathbf{U}\otimes\mathcal{I}_{n_y} \\ \mathbf{B}\otimes\mathcal{I}_{n_y} & \mathbf{V}\otimes\mathcal{I}_{n_y}\end{bmatrix}\begin{bmatrix}h_n\mathbf{F} \\ \mathbf{y}^{[n-1]}\end{bmatrix}, \tag{5}$$

where $\mathbf{A} \in \mathbb{R}^{s\times s}, \mathbf{B} \in \mathbb{R}^{q\times s}, \mathbf{U} \in \mathbb{R}^{s\times q}$ and $\mathbf{V} \in \mathbb{R}^{q\times q}$ are coefficient matrices, $\mathbf{y}^{[n]} \in \mathbb{R}^{qn_y}$ is the vector of states at the nth time step as well as the q previous time steps, $\mathbf{Y} \in \mathbb{R}^{sn_y}$ is the vector of states at the s stages at the nth timestep , $\mathbf{F} \in \mathbb{R}^{sn_y}$ is the result of evaluating f at $\mathbf{Y}$, and $\otimes$ denotes the Kronecker product. Any linear multi-step or multi-stage method can be applied by selecting the appropriate $\mathbf{A}, \mathbf{B}, \mathbf{U}$ and $\mathbf{V}$ matrices. For example, the 4th-order Runge–Kutta method has $q = 1$ and $s = 4$ with coefficient matrices

$$\mathbf{A} = \begin{bmatrix}0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0\end{bmatrix}, \mathbf{B} = \begin{bmatrix}1/6 \\ 1/3 \\ 1/3 \\ 1/6\end{bmatrix}^T, \mathbf{U} = \begin{bmatrix}1 \\ 1 \\ 1 \\ 1\end{bmatrix}, \mathbf{V} = \begin{bmatrix}1\end{bmatrix}. \tag{6}$$

The GLM formulation's utility comes from the ability to easily implement many numerical methods by choosing appropriate values of $\mathbf{A}, \mathbf{B}, \mathbf{U}$ or $\mathbf{V}$ in (5). Currently, `Ozone` implements the Runge–Kutta family using the GLM. Thus, for the next three sections, we will focus on GLM in the context of Runge–Kutta methods where $q = 1$ and $\mathbf{A} \in \mathbb{R}^{s\times s}$, $\mathbf{B} \in \mathbb{R}^{1\times s}$, $\mathbf{U} \in \mathbb{R}^{s\times 1}$, and $\mathbf{V} \in \mathbb{R}^{1\times 1}$. We believe that the GLM–based implementation of Runge–Kutta methods provides the potential for straightforward implementation of multi-step methods in future work due to its generality compared to the Butcher tableau.

2.3 Parallelization

There has been significant work done to develop formulations to solve ODEs in a parallel-computing environment. Depending on the ODE system, the best way to parallelize the solution procedure for optimal performance can greatly differ. One can identify three approaches (Burrage, 1997): parallelism across the method, parallelism across the system, and parallelism across the steps.

Parallelism across the method involves parallel function evaluations for different parts within a method and is effective when a small number of cores are available. Exploiting the sparsity of A from the Butcher tableau is one way to apply this as it can allow independent evaluations of the right hand side for both explicit and implicit Runge–Kutta methods (Nørsett and Simonsen, 1989). DifferentialEquations.jl (Rackauckas and Nie, 2017) implements this technique by offering multi-threaded versions of some methods. Another common approach is used in predictor-corrector methods which use an implicit solver to solve for the states (Burrage, 1995). With this type of parallelism, implementation is relatively simple due to the fact that existing methods can be used; however, the potential for large-scale parallelism is low (Skeel and Tam, 1992).

Parallelism across the system involves dividing the system into multiple parts and solving them in parallel (Burrage, 1997). This technique can utilize parallel computing at a larger scale than parallelism across the method and encompasses spatial parallelization which is effective when n_y is large. Two instances of this are the popular waveform relaxation and multirate methods where multiple ODEs are integrated at once. Waveform relaxation involves decoupling the ODE into several independent systems to converge to a solution and can lead to large speed ups (Bellen and Zennaro, 1993). Three formulations include the Jacobi, Gauss–Seidel and successive over-relaxation methods. Further, Sand and Burrage presents a Jacobi waveform relaxation technique where a fixed number of iterations are needed (1998). Also, Janssen and Vandewalle found promising results for the convolution successive over-relaxation method (1997).

Finally, parallelism across the steps involves parallelization of the time dimension. These techniques are not as widely used as parallelism across the system. Bellen and Zennaro use multiple shooting to allow parallelization across multiple intervals of time (1989). The parareal algorithm solves ODEs by combining parallel computation of accurate approximations with sequential computations of coarse approximations (Gander and Vandewalle, 2007). The multigrid reduction in time method is similar to the parareal algorithm but allows a grid hierarchy greater than two levels (Falgout et al., 2017) and has been implemented in PyMGRIT (Hahne et al., 2021). PFASST is also related to the parareal method but uses the spectral deferred corrections method (Dutt et al., 2000) for the time step procedure (Emmett and Minion, 2012). Picard iteration is a method for numerically integrating ODEs based on successive approximations. A variant of this approach is the modified Chebyshev-Picard iteration method which allows parallel evaluations of the ODE function by exploiting the orthogonality property of Chebyshev polynomials (Feagin and Nacozy, 1983). Speed-ups of several orders of magnitude were found compared to ODE45 when applied to orbit propagation problems (Bai and Junkins, 2012) (Read et al., 2016). Wang et al. developed the multi-step Newton–Picard method, similar to the modified Chebyshev–Picard iteration but using integral equations for better stability (Wang et al., 2020). Finally, the collocation method is a well known method to solve optimal control problems. This method discretizes the differential equation and transforms it into a nonlinear programming problem (Tsang et al., 1975).

2.4 Sensitivity Analysis

As mentioned in the introduction, gradient-based optimization requires derivatives of output variables with respect to input variables. When ODEs are embedded in a larger model, computing such derivatives often requires computing derivatives of integrated states with respect to the initial conditions, step size and static/dynamic parameters.

There has been significant attention to the development of derivative propagation methods through time integrators. A simple approach is to use a finite-difference approximation. However, it can be computationally inefficient if the number of input parameters is large. Further, inaccuracies can accumulate from subtractive cancellation and truncation error (Martins and Hwang, 2013). For optimal control problems, the sparsity of the Jacobian can be exploited to reduce the

computational cost of the finite-difference step. GPOPS-II employs sparse finite-difference methods to efficiently evaluate first and second order derivatives (Patterson and Rao, 2014). Dymos allows use of efficient finite-difference and complex-step methods (Falck et al., 2021) using OpenMDAO's coloring algorithm (Gray et al., 2019).

For better accuracy, automatic differentiation can be applied directly to the integration procedure (Griewank and Walther, 2008). This method can be difficult to implement as significant changes to the source code may be required to enable automatic differentiation. Furthermore, it can be expensive in terms of memory usage if all intermediate values must be stored. The approach used by dolfin-adjoint (Farrell et al., 2012) also utilizes automatic differentiation but at a higher level of granularity, working with symbolic representations as opposed to low-level code.

Two popular applications of automatic differentiation are the direct (also known as tangent linear) and adjoint methods. The computational cost of the direct method scales with the number of independent variables while the cost of the adjoint method increases with the number of dependent variables. Further, the continuous and discrete versions correspond to applying the two methods before and after discretization respectively (Nadarajah and Jameson, 2000). DENSERKS (Alexe and Sandu, 2009), and CVODES (Serban and Hindmarsh, 2003) implement continuous adjoint and continuous direct methods. On the other hand, FATODE (Zhang and Sandu, 2014) utilizes the discrete adjoint and discrete direct methods.

$$(remove) \tag{7}$$

For optimization problems, there are often fewer dependent variables than independent variables (Gray et al., 2019). This naturally leads to the adjoint method being generally more suitable than the direct method. The adjoint method is given by the equation

$$\frac{d\mathbf{v}}{d\mathbf{w}} = \frac{\partial \mathbf{V}}{\partial \mathbf{w}} + \boldsymbol{\Psi}^T \frac{\partial \mathbf{R}}{\partial \mathbf{w}}, \tag{8}$$

$$-\frac{\partial \mathbf{R}}{\partial \mathbf{r}}^T \boldsymbol{\Psi} = \frac{\partial \mathbf{V}}{\partial \mathbf{r}}^T, \tag{9}$$

where $\mathbf{v}$ and $\mathbf{w}$ consist of vectors of dependent and independent variables respectively while $\mathbf{V}$ is the function to compute $\mathbf{v}$. $\mathbf{r}$ and $\mathbf{R}$ defines vectors of intermediate state variables and their functions as residuals. $\boldsymbol{\Psi}$ is called the adjoint matrix and is defined from the linear system in Equation (9). Solving for the adjoint matrix is the main computational bottleneck as it requires solving a linear system for each element in the vector $\mathbf{v}$. Hence, minimizing the size of v is generally desirable to reduce the number of systems to solve. Alternative approaches, such as reformulating Equation (9) as a fixed point system, can also be utilized when possible (Christianson, 1994).

For sequential systems like time-integration procedures, the upper block triangular structure of the transposed Jacobian matrix $\frac{\partial R}{\partial r}^T$ can be exploited, allowing Ψ to be computed using back substitution.. Due to the use of back substitution, the adjoint method requires completion of the model evaluation before operation. Additionally, the partial derivatives within each Jacobian must be known and computed before solving Equation (8) and (9).

A common issue with the adjoint method is the high memory cost. A direct implementation of the adjoint method requires full allocation of all intermediate values as equations (8) and (9) are solved after model evaluation. The direct method does not suffer from this limitation as it can be solved concurrently with the model evaluation which allows values to be discarded after they are used. One way to address this issue with the adjoint method is to use the checkpointing technique.

Checkpointing consists of saving intermediate state values at specific checkpoints in the forward procedure and reevaluating unsaved states when needed in the adjoint computation (Griewank and Walther, 2000). Because only states from specific points in time are kept in memory as opposed to the full time history, the total memory required is reduced. Many algorithms have been developed to determine efficient checkpoint locations. For example, Griewank and Walther determined an optimal binomial checkpointing scheme that minimizes the number of recalculations (2000). The software project dolfin-adjoint uses this approach for their sensitivity analysis (Farrell et al., 2012). Wang et al. presents a dynamic checkpointing algorithm for problems where the number of states are not known in advance (2009). Further, Aupy et al. designed an optimal algorithm for two levels of checkpoints: writing/reading to memory or disk (2016). Another method with a

more straightforward implementation is to simply arrange checkpoints with a uniform distribution across the time span. This simpler approach is used by CVODES (Serban and Hindmarsh, 2003). Checkpointing is not limited to time loops or pseudo-time loops; it is a general technique that can be applied in reverse-mode automatic differentiation to reduce memory overhead in various contexts.

2.5 System Modeling Methodology

As mentioned in Sect. 1, one of the main differences between Ozone and its predecessor (presented by Hwang and Munster (2018)) is Ozone's implementation in a new system modeling framework called the Computational System Design Language (CSDL) (Gandarillas et al., 2024). The predecessor was also named Ozone but was never released with a version number and was written for the OpenMDAO modeling framework (Gray et al., 2019). CSDL is a novel algebraic modeling language in Python that uses a graph-based approach to construct user-defined computational models.

A graph representation of the model enables automatic sensitivity analysis and efficient model execution while still providing a Pythonic interface for the user. To define models, CSDL offers a NumPy-like API, allowing users to easily assemble complex optimization models and interface submodels together. CSDL dynamically records mathematical dependencies between variables and operations as the model is defined, creating a graph data structure of the user's code.

Automatic adjoint-based sensitivities are then calculated by analyzing the dependency relationships between dependent and independent variables in the graph. Further, efficient model evaluation is enabled by translating the graph to different computational backends. CSDL currently supports a JAX (Bradbury et al., 2018) backend, providing significant speedups compared to native Python. We refer readers to the CSDL documentation linked in Ozone's repository for further information.

For Ozone, the use of CSDL offers key benefits. First, ODE functions can be defined without manually implementing derivatives, reducing model development time. Second, the computation time of solving the ODE and optimization problem is significantly lower than if they were solved in native Python. These features are key advantages of CSDL not present in OpenMDAO Gray et al. (2019).

3 Methodology

The Ozone library currently implements any method from the implicit and explicit Runge–Kutta family using the general linear method (GLM) formulation as described in Sect. 2.1 and 2.2. The methods are solved using one of four approaches. Two of the approaches are sequential in time: time-marching and time-marching with checkpointing. The other two approaches are in the parallelism across the steps family as described in Sect. 2.3: Picard iteration and collocation. Although the two prior methods have the capability to be executed in a parallel computing environment, Ozone currently implements a vectorized version, keeping the parallelization for future work. Derivative computation of the integration procedure is performed using the adjoint method as per Sect. 2.4. Finally, Ozone is compatible with CSDL, as described in Sect. 2.5, which enables automatic derivative computation.

Using the GLM equations, we will now develop the theory and implementation of the four introduced approaches. A general overview of the solution approaches can be seen in Figure 1. Table 1 shows a high level comparison between approaches.

3.1 Time-Marching (with checkpointing)

3.1.1 Forward Integration

In Sect. 2.2, we discussed the GLM equations and their relation to the Runge–Kutta methods. We now describe how they are used for the time-marching and time-marching with checkpointing

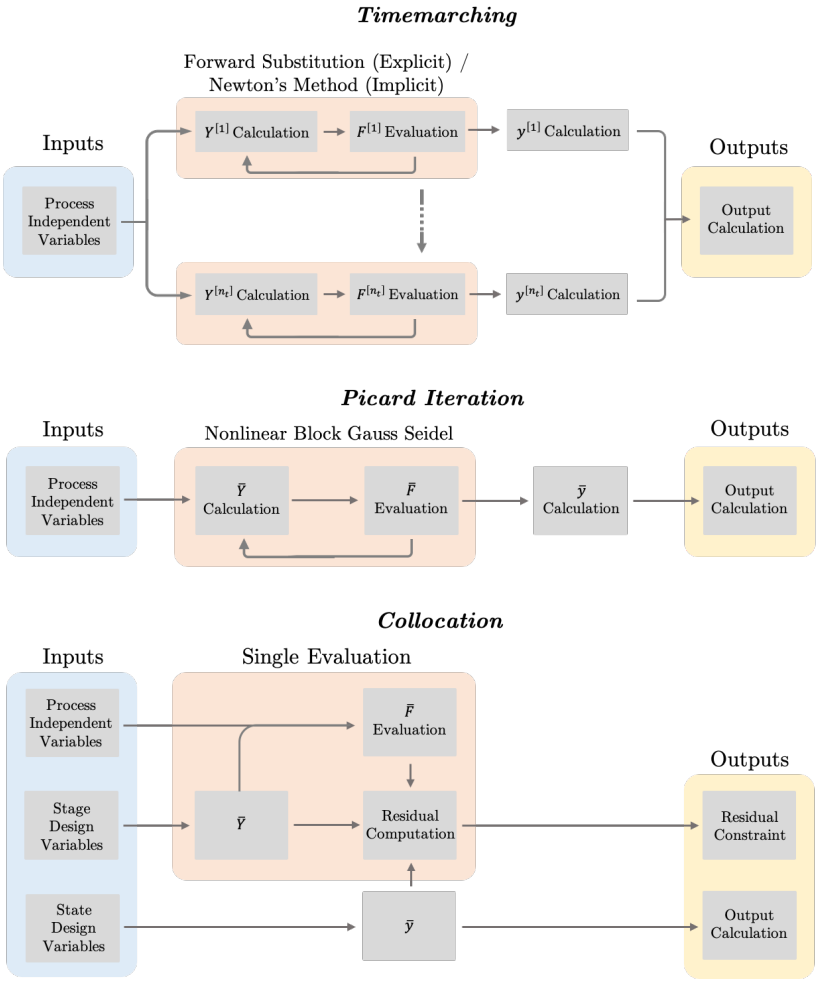

Fig. 1 Comparison of the time-marching, time-marching with checkpointing, Picard iteration, and collocation solution approaches. The time-marching and time-marching with checkpointing approaches sequentially solve for the states one time step at a time. On the other hand, Picard iteration and collocation approaches solve for the full state history vector in parallel where concatenated vectors $\bar{\mathbf{F}}$, $\bar{\mathbf{Y}}$ and $\bar{\mathbf{y}}$ are defined in Equation (31).

approach. To restate, the objective is to solve for the state history $\mathbf{y}^{[n]}$ for $n = 0, \ldots, n_t$. Sequential computation of the states $\mathbf{y}^{[n]}$ from Equation (5) is straightforward. At each time step n, the evaluated stages $\mathbf{F}^{[n]}$ are calculated from

$$\mathbf{Y}^{[n]} = \tilde{\mathbf{A}} h_n \mathbf{F}^{[n]} + \tilde{\mathbf{U}} \mathbf{y}^{[n-1]}, \qquad \mathbf{F}^{[n]} = F(\mathbf{Y}^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s), \tag{10}$$

where $\tilde{\mathbf{A}} = \mathbf{A} \otimes \mathcal{I}_{n_y}$ with identical definitions for $\tilde{\mathbf{B}}, \tilde{\mathbf{U}}$ and $\tilde{\mathbf{V}}$. $\mathbf{P}_d^{[n]}$ denotes dynamic parameters linearly interpolated across s stages for time step n and $\mathbf{p}_s$ denotes constant static parameters. To solve for $\mathbf{F}^{[n]}$, Newton's method is used to first compute $\mathbf{Y}^{[n]}$:

$$\mathbf{Y}_{k+1}^{[n]} = \mathbf{Y}_k^{[n]} - \left(\mathcal{I} - \tilde{\mathbf{A}} h_n \frac{\partial F(\mathbf{Y}_k^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s)}{\partial \mathbf{Y}} \right)^{-1} \left(\mathbf{Y}_k^{[n]} - \tilde{\mathbf{A}} h_n F(\mathbf{Y}_k^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s) - \tilde{\mathbf{U}} \mathbf{y}^{[n-1]} \right) \tag{11}$$

where the iterations terminate when the convergence criteria is met. Then, $\mathbf{F}^{[n]}$ is computed from Equation (10) using $\mathbf{Y}^{[n]}$.

If $\tilde{\mathbf{A}}$ is strictly lower triangular as is the case for explicit Runge–Kutta methods (see Equation (6)), $\mathbf{Y}^{[n]}$ is solved using forward substitution as opposed to Newton's method. The forward substitution algorithm is implemented without explicit construction of the full linear system and is preferred over Newton's method due to the computational efficiency. The state solution $\mathbf{y}^{[n]}$ is

Table 1 A high-level comparison of the four approaches implemented in Ozone

	Time-marching	**Time-marching with checkpointing**	**Picard iteration**	**Collocation**
Description	Standard sequential technique	Identical to time-marching but with checkpointing	Parallel integration using fixed-point iteration	Parallel integration as optimization problem
Primary strength	Well studied technique	Low memory costs	Parallel derivative free integration	Does not require full ODE solution at every optimization iteration
Primary weakness	Little parallelization	Slow derivative computation	Instability and high memory costs	Additional complexity to optimization problem
Memory cost	Medium	Low	High	High
Ability to be parallelized	None (Low if implicit)	None (Low if implicit)	High	High
ODE nonlinearity impact on performance	None	None	High	High
Requires solution to implicit system	No (Yes if implicit)	No (Yes if implicit)	Yes	No
Example	Van Der Pol Oscillator (5.2)	Nonlinear Reaction-Diffusion Process (5.4)	Lunar Vehicle Ascent (5.1)	Trajectory Optimization (5.3)

then computed from $\mathbf{F}^{[n]}$ in one procedure through

$$\mathbf{y}^{[n]} = \tilde{\mathbf{B}} h_n \mathbf{F}^{[n]} + \tilde{\mathbf{V}} \mathbf{y}^{[n-1]}. \tag{12}$$

The final step for the integration procedure for time-marching is to repeat the computations from Equations (10) through (12) to sequentially compute $\mathbf{y}^{[n]}$ as a function of $\mathbf{y}^{[n-1]}$ for $n = 1, \ldots, n_t$. Values for all time steps $n = 0, \ldots, n_t$ are stored for reuse in the computation of the adjoint method.

For time-marching with checkpointing, the integration procedure is identical. However, to reduce the memory footprint, the state solution is not stored at every time step. Instead, the checkpointing scheme selectively stores states for only a subset of time steps. A limitation of this procedure is that the full state history is not available after the forward integration is complete, meaning some state information is lost. We introduce two new output types from the time integrator that are compatible with this restriction: profile outputs and field outputs. Profile outputs and field outputs contain state information from all time steps without the need to store the states for every time step. We now describe both types of outputs.

A profile output $\mathbf{J}_p \in \mathbb{R}^{(n_t+1)\times n_p}$ is defined as

$$\mathbf{J}_p = \begin{bmatrix} \mathbf{J}_p^{[0]} \\ \vdots \\ \mathbf{J}_p^{[n_t]} \end{bmatrix} = \begin{bmatrix} f_p(\mathbf{y}^{[0]}, \mathbf{p}_d^{[0]}, \mathbf{p}_s) \\ \vdots \\ f_p(\mathbf{y}^{[n_t]}, \mathbf{p}_d^{[n_t]}, \mathbf{p}_s) \end{bmatrix}, \tag{13}$$

where $f_p : \mathbb{R}^{n_y+n_{pd}+n_{ps}} \to \mathbb{R}^{n_p}$ is a function defined by the user and $n_p \leq n_y$. Because partial state information is stored at every time step, the time profile is preserved. Unlike the full state history, $\mathbf{J}_p^{[n]}$ is kept in memory for all $0 \leq n \leq n_t$ by computing $f_p(\mathbf{y}^{[n]}, \mathbf{p}_d^{[n]}, \mathbf{p}_s)$ every time the state solution is solved in Equation (12). An example of a realistic application of a profile output in an optimization problem is a constraint on the spatial average of a PDE at every time step. For such a problem, the profile output function is $f_p(\mathbf{y}^{[n]}, \mathbf{p}_d^{[n]}, \mathbf{p}_s) = \text{average}(\mathbf{y}^{[n]})$. In this context, $n_p = 1$ and the size of the final output $\mathbf{J}_p \in \mathbb{R}^{(n_t+1)\times 1}$ is much smaller than the size of the full

state history $n_t n_y$. Note that setting the full state solution history as an output can be generalized as a profile output by:

$$\mathbf{J}_p = \begin{bmatrix} \mathbf{y}^{[0]} \\ \vdots \\ \mathbf{y}^{[n_t]} \end{bmatrix} = \begin{bmatrix} f_p(\mathbf{y}^{[0]}, \mathbf{p}_d^{[0]}, \mathbf{p}_s) \\ \vdots \\ f_p(\mathbf{y}^{[n_t]}, \mathbf{p}_d^{[n_t]}, \mathbf{p}_s) \end{bmatrix}. \tag{14}$$

The size reduction of profile outputs compared to the full set of states is n_y/n_p. As long as f_p is defined such that $n_p << n_y$, the memory requirements are substantially decreased.

A field output $\mathbf{J}_f \in \mathbb{R}^{n_y}$ is defined as a linear combination of states across time by

$$\mathbf{J}_f = \sum_{n=0}^{n_t} \alpha_n \mathbf{y}^{[n]}, \tag{15}$$

where α_n are weighting coefficients defined by the user. Unlike profile outputs, field outputs preserve the size of the state. $\mathbf{J}_f$ is generated by computing $\mathbf{J}_f^n = \alpha_n \mathbf{y}^{[n]} + \mathbf{J}_f^{n-1}$ at every timestep where $\mathbf{J}_f^0 = \alpha_0 \mathbf{y}_0$ and $\mathbf{J}_f$ is $\mathbf{J}_f^{n_t}$. . A possible application of a field output in an optimization problem is a constraint on the temporal average of a PDE. This is achieved by setting the coefficients to be $\alpha_n = 1/(n_t + 1)$. Note that having the state of the kth time step $\mathbf{y}^{[k]}$ as an integrator output could be generalized as a field output through the coefficients $\alpha_k = 1$, $\alpha_{i \neq k} = 0$. Field outputs reduce the output size by n_t compared to the full state history. For problems where $n_t >> 1$, the reduction is significant.

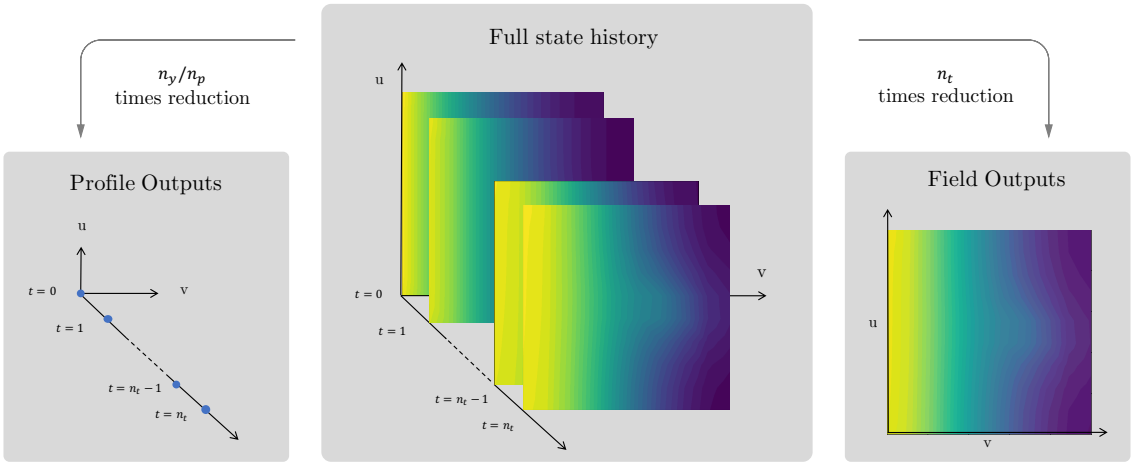

Fig. 2 Profile and field outputs and how their sizes relate to the full state history. $n_y = n_u n_v$ is the total number of state variables where n_u and n_v is the number of discretizations in the u and v dimensions respectively. n_p is the output size of the user defined f_p function.

To illustrate both profile outputs and field outputs by example, Figure (2) shows the size of profile outputs and field outputs compared to the full state on the 2D heat equation problem. As mentioned previously, the spatial and temporal average could be the profile and field output respectively.

3.1.2 Derivative Computation

As mentioned in Sect. 2.4, derivatives are computed with the adjoint method using a matrix-free approach. To review, the dependent variables consist of the recently introduced profile outputs $\mathbf{J}_p$ and field outputs $\mathbf{J}_f$ while the independent variables include initial conditions $\mathbf{y}_0$, dynamic parameters $\bar{\mathbf{p}}_d = [\mathbf{p}_d^{[0]T}, \ldots, \mathbf{p}_d^{[n_t]T}]^T$, static parameters $\mathbf{p}_s$, and the time step vector $\mathbf{H} \in \mathbb{R}^{n_t}$ defined as $[h_1, \ldots, h_{n_t}]^T$.

We use Equation (8) and (9) and its nomenclature to derive the adjoint derivative calculation. The matrix-free adjoint method from Equation (8) for time-marching yields

$$\boldsymbol{\upsilon}^T \frac{d(\mathbf{J}_f, \mathbf{J}_p)}{d(\mathbf{y}_0, \bar{\mathbf{p}}_d, \mathbf{p}_s, \mathbf{H})} = \boldsymbol{\upsilon}^T \frac{\partial(F_{J_f}, F_{J_p})}{\partial(\mathbf{y}_0, \bar{\mathbf{p}}_d, \mathbf{p}_s, \mathbf{H})} + \tilde{\boldsymbol{\Psi}}^T \frac{\partial R}{\partial(\mathbf{y}_0, \bar{\mathbf{p}}_d, \mathbf{p}_s, \mathbf{H})}, \tag{16}$$

$$-\frac{\partial R}{\partial \mathbf{r}}^T \tilde{\boldsymbol{\Psi}} = \frac{\partial(F_{J_f}, F_{J_p})}{\partial \mathbf{r}}^T \boldsymbol{\upsilon}, \tag{17}$$

where $r \in \mathbb{R}^{(2s+1)n_y n_t}$ and R is a concatenation of the three computations from Equations (10) and (12) as well as their outputs $\mathbf{Y}^{[n]}$, $\mathbf{F}^{[n]}$ and $\mathbf{y}^{[n]}$ for time steps $n = 1, \dots n_t$ generalized as residuals. For generality, we assume that both $\mathbf{J}_f$ and $\mathbf{J}_p$ are outputs to the integrator and $\mathbf{y}_0 \in \mathbb{R}^{n_y}$, $\bar{\mathbf{p}}_d \in \mathbb{R}^{(n_t+1)\times n_{pd}}$, $\mathbf{p}_s \in \mathbb{R}^{n_{ps}}$, and $\mathbf{H} \in \mathbb{R}^{n_t}$ are inputs. Finally, F_{J_f} and F_{J_p} represent the calculations of the outputs $\mathbf{J}_f$ and $\mathbf{J}_p$.

We implement a matrix-free method to avoid construction of the full matrices and to limit the number of linear systems to solve. Equation (17) has the following structure when applied to the time-marching scheme:

$$-\frac{\partial R}{\partial \mathbf{r}}^T = -\begin{bmatrix} \frac{\partial R}{\partial \mathbf{r}}^T_{1,1} & \frac{\partial R}{\partial \mathbf{r}}^T_{1,2} & \cdots & 0 & 0 \\ 0 & \frac{\partial R}{\partial \mathbf{r}}^T_{2,2} & & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & & \frac{\partial R}{\partial \mathbf{r}}^T_{n_t-1,n_t-1} & \frac{\partial R}{\partial \mathbf{r}}^T_{n_t-1,n_t} \\ 0 & 0 & \cdots & 0 & \frac{\partial R}{\partial \mathbf{r}}^T_{n_t,n_t} \end{bmatrix}, \tag{18}$$

$$\tilde{\boldsymbol{\Psi}} = \begin{bmatrix} \tilde{\boldsymbol{\Psi}}^{[1]} \\ \tilde{\boldsymbol{\Psi}}^{[2]} \\ \vdots \\ \tilde{\boldsymbol{\Psi}}^{[n_t-1]} \\ \tilde{\boldsymbol{\Psi}}^{[n_t]} \end{bmatrix}, \qquad \frac{\partial(F_{J_f}, F_{J_p})}{\partial \mathbf{r}}^T \boldsymbol{\upsilon} = \begin{bmatrix} \mathbf{R}_{in}^{[1]} \\ \mathbf{R}_{in}^{[2]} \\ \vdots \\ \mathbf{R}_{in}^{[n_t-1]} \\ \mathbf{R}_{in}^{[n_t]} \end{bmatrix}, \tag{19}$$

where

$$\frac{\partial R}{\partial \mathbf{r}}^T_{n,n} = \begin{bmatrix} \left(\mathcal{I} - \tilde{\mathbf{A}} h_n \frac{\partial F(\mathbf{Y}^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s)}{\partial \mathbf{Y}}\right)^T & \frac{\partial F(\mathbf{Y}^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s)}{\partial \mathbf{Y}}^T & 0 \\ 0 & \mathcal{I} & h_n \tilde{\mathbf{B}}^T \\ 0 & 0 & \mathcal{I} \end{bmatrix}, \tag{20}$$

$$\frac{\partial R}{\partial \mathbf{r}}^T_{n,n+1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\tilde{\mathbf{U}}^T & 0 & -\tilde{\mathbf{V}}^T \end{bmatrix}, \qquad \mathbf{R}_{in}^{[n]} = \begin{bmatrix} 0 \\ 0 \\ \frac{\partial f_p(\mathbf{y}^{[n]}, \mathbf{p}_d^{[n]}, \mathbf{p}_s)}{\partial \mathbf{y}}^T \boldsymbol{\upsilon}_p^{[n]} + \alpha_n \boldsymbol{\upsilon}_f \end{bmatrix}, \tag{21}$$

for $1 \leq n \leq n_t$ where $\boldsymbol{\upsilon}_p^{[n]}$ and $\boldsymbol{\upsilon}_f$ are the given propagated derivative vector from the upstream vector Jacobian product corresponding to the profile output of the nth time step and the field output respectively. To solve for $\tilde{\boldsymbol{\Psi}}^{[n]}$, we apply back substitution to obtain

$$\tilde{\boldsymbol{\Psi}}_c^{[n]} = -\frac{\partial f_p(\mathbf{y}^{[n]}, \mathbf{p}_d^{[n]}, \mathbf{p}_s)}{\partial \mathbf{y}}^T \boldsymbol{\upsilon}_p^{[n]} - \alpha_n \boldsymbol{\upsilon}_f + \tilde{\mathbf{U}}^T \tilde{\boldsymbol{\Psi}}_a^{[n+1]} + \tilde{\mathbf{V}}^T \tilde{\boldsymbol{\Psi}}_c^{[n+1]} \tag{22}$$

$$\tilde{\boldsymbol{\Psi}}_b^{[n]} = h_n \tilde{\mathbf{B}}^T \tilde{\boldsymbol{\Psi}}_c^{[n]}, \qquad -\left(\mathcal{I} - \tilde{\mathbf{A}} h_n \frac{\partial F(\mathbf{Y}^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s)}{\partial \mathbf{Y}}\right)^T \tilde{\boldsymbol{\Psi}}_a^{[n]} = \frac{\partial F(\mathbf{Y}^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s)}{\partial \mathbf{Y}}^T \tilde{\boldsymbol{\Psi}}_b^{[n]}, \tag{23}$$

where $\tilde{\boldsymbol{\Psi}}^{[n]} = [\tilde{\boldsymbol{\Psi}}_a^{[n]T}, \tilde{\boldsymbol{\Psi}}_b^{[n]T}, \tilde{\boldsymbol{\Psi}}_c^{[n]T}]^T$. The final step is to repeat the calculations iteratively from $n = n_t$ to $n = 1$ with initial conditions $\tilde{\boldsymbol{\Psi}}_c^{[n_t]} = 0$ and $\tilde{\boldsymbol{\Psi}}_a^{[n_t]} = 0$. This allows full computation of the adjoint vector $\tilde{\boldsymbol{\Psi}}$ by stepping backwards through the integration time steps. The significance

of this formulation is that construction of the full matrix is completely avoided and the adjoint vector can be solved sequentially one time step at a time. Further, we exploit the upper block diagonal structure in Equation (18) to solve the full linear system using backwards substitution. The decomposed system now requires a smaller linear system to be solved at every time step as shown in Equation (23) compared to solving Equation (18) naively. If an explicit Runge–Kutta method is specified, Equation (23) is solved by backward substitution as $\tilde{\mathbf{A}}^T$ is upper triangular. Otherwise, an iterative solver is used.

Building the full $\tilde{\mathbf{\Psi}}$ vector from the iterative procedure above and then solving Equation (16) requires a significant amount of memory. To address this, a similar partitioning procedure to the one above is used to allow the computation of Equation (16) without building $\tilde{\mathbf{\Psi}}$ in full. This also comes from exploiting the matrix structures of the partial Jacobian matrices.

The matrix structure of the partial Jacobian can be partitioned columnwise into the respective parts for each independent variable: the initial conditions, dynamic parameters, static parameters, and the time step vector. Further, they can be partitioned further row-wise for each time step of the ODE. Finally, exploiting the triangular structure again results in the equations

$$\boldsymbol{\upsilon}^T \frac{d\mathbf{J}}{d\mathbf{y}_0} = \sum_{n=1}^{N} \left(-\tilde{\mathbf{\Psi}}_a^{[n]} \tilde{\mathbf{U}} - \tilde{\mathbf{\Psi}}_c^{[n]} \tilde{\mathbf{V}} \right) + {\boldsymbol{\upsilon}_p^{[0]}}^T \frac{\partial f_p(\mathbf{y}_0, \mathbf{p}_d^{[0]}, \mathbf{p}_s)}{\partial \mathbf{y}} + \boldsymbol{\upsilon}_f \alpha_0,$$

$$\boldsymbol{\upsilon}^T \frac{d\mathbf{J}}{d\mathbf{p}} = \sum_{n=1}^{N} \left(-\tilde{\mathbf{\Psi}}_a^{[n]} \tilde{\mathbf{A}} \frac{\partial F(\mathbf{Y}^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s)}{\partial p} - \tilde{\mathbf{\Psi}}_b^{[n]} \frac{\partial F(\mathbf{Y}^{[n]}, \mathbf{P}_d^{[n]}, \mathbf{p}_s)}{\partial p} \right) \tag{24}$$

$$\boldsymbol{\upsilon}^T \frac{d\mathbf{J}}{d\mathbf{H}} = \sum_{n=1}^{N} \left(-\tilde{\mathbf{\Psi}}_a^{[n]} \tilde{\mathbf{A}} \mathbf{F}^{[n]} - \tilde{\mathbf{\Psi}}_c^{[n]} \tilde{\mathbf{B}} \mathbf{F}^{[n]} \right), \tag{25}$$

where $\mathbf{p}^{[n]}$ is the concatenation of static parameters $\mathbf{p}_s$ and dynamic parameters $\mathbf{p}_d^{[n]}$. Evaluation of the right side can occur incrementally by storing a running total of the summation term as Equations (22) to (23) are computed. Thus, the full matrices and adjoint vector as stated in Equation (18) are never formed in full. After the preceding equations are computed for $1 \leq n \leq n_t$, the time-independent middle term in Equation (16) is computed and added to the vector Jacobian products. The resulting value is the vector Jacobian products for the initial conditions, parameters and time step vector. We note that the derivation of the vector Jacobian products via accumulation of adjoint vectors in the reverse direction of the integration, as outlined in Equations (22) through (25), reproduces established methods in the literature (Hager, 2000; Lions, 1971).

The preceding equations of the adjoint method are similar for both time-marching and time-marching with checkpointing approaches. Whenever the adjoint method procedure is called using the time-marching approach, $\mathbf{F}^{[n]}$ and its derivatives, $\mathbf{Y}^{[n]}$ and $\mathbf{y}^{[n]}$ are needed for $1 \leq n \leq n_t$. If n_y and n_t is small enough, all of the preceding values can be stored in memory when the ODE is integrated.

An advantage of this approach is that the cost of computing the vector Jacobian product is independent of those of the ODE function and its derivatives. When the number of vector Jacobian products is large (for example when there are path constraints), this approach can save significant computational cost. However, an obvious disadvantage is the expensive memory costs. If n_y and n_t are large, storing states $\mathbf{F}^{[n]}$, $\mathbf{Y}^{[n]}$ and $\mathbf{y}^{[n]}$ and especially the partial derivatives of $\mathbf{F}^{[n]}$ for all time steps may be infeasible. The estimated number of values stored using the time-marching approach is given by

$$m_t = n_t n_y (s+1) + n_t n_y^2 (s+1). \tag{26}$$

A solution to this problem is to use the aforementioned checkpointing scheme as done by the time-marching with checkpointing approach. During the ODE integration process, the state history is not stored for all time steps. Instead, state information is saved only at each checkpoint i_n, where $i_n = n * round(n_t/n_c)$ is the nth checkpoint and n_c is the number of checkpoints. In short, state information is saved every n_t/n_c steps as opposed to every step. When derivative information is needed, the interval i_n to i_{n-1} is first integrated using a hot restart from checkpoint i_n with all state information stored in memory. Then, the adjoint method is computed using the stored state information from i_{n-1} to i_n. This process is repeated for $1 \leq n \leq n_c$ until the full

vector Jacobian product is built. Figure 3 shows this procedure applied to six checkpoints. An advantage of this approach is that computing the vector Jacobian product is independent of the computational cost of the ODE function and its derivatives. When the number of vector Jacobian products is large (for example, when there are path constraints), this approach can save significant computational cost. However, an obvious disadvantage is the expensive memory costs. If n_y and n_t are large, storing intermediate values such as $\mathbf{F}^{[n]}$, $\mathbf{Y}^{[n]}$ and $\mathbf{y}^{[n]}$ and especially the partial derivatives of $\mathbf{F}^{[n]}$ for all time steps may be infeasible. The estimated maximum number of values stored using the time-marching approach with an implicit method is given by

$$m_t = n_t n_y (2s + 1) + n_t (n_y s)^2, \tag{27}$$

where we assume the ODE function to be linear without any input parameters.

To reduce maximum memory usage, the aforementioned checkpointing scheme is utilized by the time-marching with checkpointing approach. During the ODE integration process, the state history is not stored for all time steps. Instead, state information is saved only at each checkpoint i_n, where $i_n = n * round(n_t/n_c)$ is the nth checkpoint and n_c is the number of checkpoints. In short, state information is saved every n_t/n_c steps as opposed to every step. When derivative information is needed, the interval i_n to i_{n-1} is first integrated using a hot restart from checkpoint i_n with all state information stored in memory. Then, the adjoint method is computed using the stored state information from i_{n-1} to i_n. This process is repeated for $1 \leq n \leq n_c$ until the full vector Jacobian product is built. Figure 3 shows this procedure applied to six checkpoints.

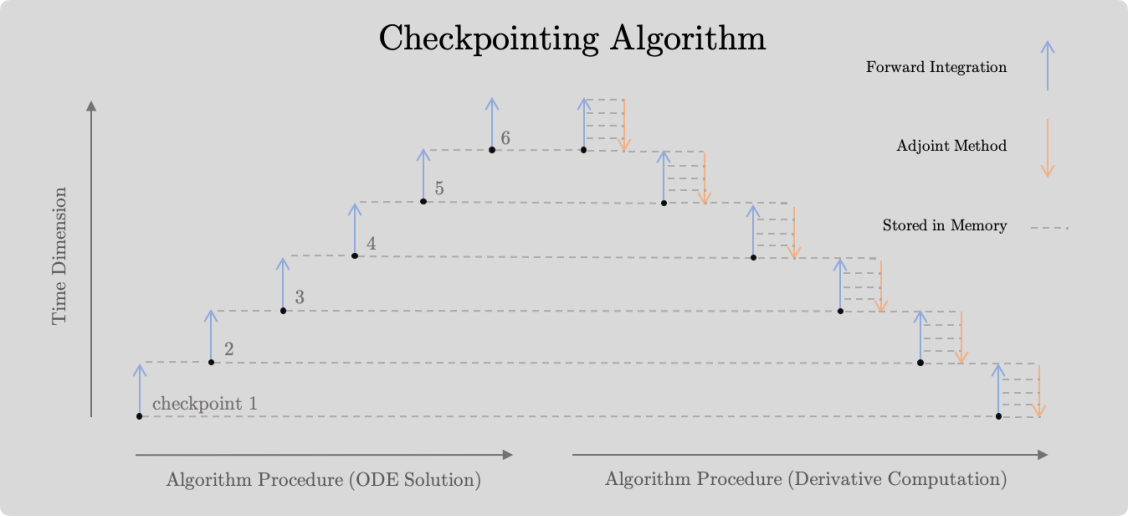

Fig. 3 An overview of the checkpointing algorithm implemented in Ozone. The horizontal axis represents the sequence of steps followed by the algorithm to solve the ODE and compute its derivatives. The vertical axis denotes the current time step of the ODE temporal dimension being processed by the algorithm. A dotted line represents a state stored in memory at a certain point of the algorithm procedure. Note that the maximum memory consumption occurs during the derivative calculation.

The value of n_c has an effect on the amount of memory required as a high value of n_c requires many checkpoints to store while a low value of n_c means the interval to store state information is long. The estimated maximum number of values stored by the checkpointing approach is approximated by

$$m_c = n_c n_y + \frac{n_t}{n_c} n_y + (n_y s)^2, \tag{28}$$

where we make the same assumptions as Equation (27). Hence, m_c is minimized when $n_c = \sqrt{n_t}$ as shown in Figure 4. In this figure, a sample ODE with a large number of states and time steps was solved along with its derivatives. Maximum memory was measured using four different Python packages to validate measurements. Equation (27) is an estimate on the theoretical maximum

number of values stored, not practical memory usage on a computer. Thus, Figure 4 shows slight differences between the theoretical and measurement curve. The inconsistencies can be attributed to overhead memory costs not taken into account in Equation (28). Despite the differences, the calculated optimal number of checkpoints is consistent with the measured values. `Ozone` allows use

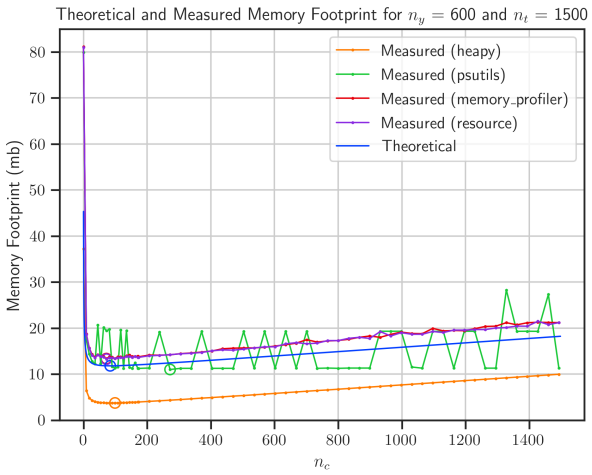

Fig. 4 Maximum allocated memory as a function of n_c for a sample ODE. The circle denotes the optimal number of checkpoints for the respective memory measurement method.

of both the time-marching approach of storing the full integration for reuse in the adjoint method as well as the time-marching with checkpointing approach that recomputes state information for the adjoint method.

3.2 Picard Iteration

The Picard iteration approach solves for the entire state solution at once by vectorizing the GLM formulation and using a nonlinear solver. Because the solution history is solved all at once, the integration procedure is no longer sequential and allows parallelization or vectorization across time. However, a disadvantage of this approach is the memory requirement of storing the full state history. The nonlinear system is formed by concatenating Equation (5) across all time steps to obtain

$$\begin{bmatrix} \bar{\mathbf{Y}} \\ \bar{\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{A}} & \bar{\mathbf{U}} \\ \bar{\mathbf{B}} & \bar{\mathbf{V}} \end{bmatrix} \begin{bmatrix} \bar{\mathbf{H}}\bar{\mathbf{F}} \\ \bar{\mathbf{y}} \end{bmatrix}, \tag{29}$$

where block diagonal matrices $\bar{\mathbf{A}}$, $\bar{\mathbf{B}}$, $\bar{\mathbf{U}}$ and $\bar{\mathbf{V}}$ are

$$\bar{\mathbf{A}} = \begin{bmatrix} \tilde{\mathbf{A}} & 0 & \dots & 0 \\ 0 & \tilde{\mathbf{A}} & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & \tilde{\mathbf{A}} \end{bmatrix}, \bar{\mathbf{B}} = \begin{bmatrix} 0 & 0 & \dots & 0 \\ \tilde{\mathbf{B}} & 0 & \dots & 0 \\ 0 & \tilde{\mathbf{B}} & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & \tilde{\mathbf{B}} \end{bmatrix}, \bar{\mathbf{U}} = \begin{bmatrix} \tilde{\mathbf{U}} & 0 & \dots & 0 & 0 \\ 0 & \tilde{\mathbf{U}} & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & \tilde{\mathbf{U}} & 0 \end{bmatrix}, \bar{\mathbf{V}} = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ \tilde{\mathbf{V}} & 0 & \dots & 0 & 0 \\ 0 & \tilde{\mathbf{V}} & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & \tilde{\mathbf{V}} & 0 \end{bmatrix}. \tag{30}$$

These are simply $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{U}}, \tilde{\mathbf{V}}$ expanded across all time steps. The vectors $\bar{\mathbf{Y}}$, $\bar{\mathbf{F}}$ and $\bar{\mathbf{y}}$ are concatenations across time steps where $\bar{\mathbf{y}}$ includes the initial conditions:

$$\bar{\mathbf{Y}} = \begin{bmatrix} \mathbf{Y}^{[1]} \\ \vdots \\ \mathbf{Y}^{[n_t]} \end{bmatrix}, \qquad \bar{\mathbf{F}} = F(\bar{\mathbf{Y}}, \bar{\mathbf{P}}_d, \mathbf{p}_s) = \begin{bmatrix} \mathbf{F}^{[1]} \\ \vdots \\ \mathbf{F}^{[n_t]} \end{bmatrix}, \qquad \bar{\mathbf{y}} = \begin{bmatrix} \mathbf{y}^{[0]} \\ \vdots \\ \mathbf{y}^{[n_t]} \end{bmatrix}. \tag{31}$$

$\mathbf{Y}$, F and $\mathbf{y}$ are identical to the definitions used in Sect. 3.1.1. We rearrange Equation (29) to get the system

$$\bar{\mathbf{Y}} - \bar{\mathbf{U}}\bar{\mathbf{y}} = \bar{\mathbf{A}}\bar{\mathbf{H}}\bar{\mathbf{F}}, \qquad (\mathcal{I} - \bar{\mathbf{V}})\bar{\mathbf{y}} = \bar{\mathbf{B}}\bar{\mathbf{H}}\bar{\mathbf{F}}. \tag{32}$$

The state vector $\bar{\mathbf{y}}$ is isolated to obtain

$$\bar{\mathbf{Y}} = \bar{\mathbf{A}}\bar{\mathbf{H}}\bar{\mathbf{F}} + \bar{\mathbf{U}}\left(\mathcal{I} - \bar{\mathbf{V}}\right)^{-1}\bar{\mathbf{B}}\bar{\mathbf{H}}\bar{\mathbf{F}} + \bar{\mathbf{U}}\left(\mathcal{I} - \bar{\mathbf{V}}\right)^{-1}\bar{\mathbf{y}_0}, \tag{33}$$

$$\bar{\mathbf{y}} = \left(\mathcal{I} - \bar{\mathbf{V}}\right)^{-1}\bar{\mathbf{B}}\bar{\mathbf{H}}\bar{\mathbf{F}} + \left(\mathcal{I} - \bar{\mathbf{V}}\right)^{-1}\bar{\mathbf{y}_0}, \tag{34}$$

where $\bar{\mathbf{y}_0}$ is $[\mathbf{y}_0^T, 0, \ldots, 0]^T$. To solve for the vector of integrated states $\bar{\mathbf{y}}$ in Equation (34), Equation (33) is implicitly solved for $\bar{\mathbf{F}}$. This is done using a nonlinear block Gauss Seidel algorithm which is simply an iteration between $\bar{\mathbf{F}}$ in Equation (31) and Equation (33) until the convergence criterion is met. The inverse of the coefficient matrix $\mathcal{I} - \bar{V}$ and the rest of the coefficient matrices are precomputed. Further, the coefficient matrices are stored in sparse format as $\bar{\mathbf{A}}, \bar{\mathbf{B}}, \bar{\mathbf{U}}, \bar{\mathbf{V}}$ are large and sparse. $\bar{\mathbf{y}}$ can be solved explicitly in one step using $\bar{\mathbf{F}}$ from Equation (34). Although often not needed, $\mathbf{J}_f$ and $\mathbf{J}_p$ can be computed using $\bar{\mathbf{y}}$.

The validity of the parallelization across time can be seen in Equation (33) as the elements of $\bar{\mathbf{F}}$, $(\mathbf{F}^{[1]}, \ldots, \mathbf{F}^{[n_t]})$ can be evaluated all at once concurrently, independent of each other.

For sensitivity analysis, we rely on CSDL's automatic differentiation capabilities. The derivatives are not automatically propagated through the iterative procedure in Equation (33). Instead, the adjoint method is used by representing the nonlinear equation as a residual (Sperry et al., 2023). Therefore, a solution of a linear system is required during the derivative computation procedure.

3.3 Collocation

The collocation approach solves for the entire state solution in a similar manner to Picard iteration. However, instead of utilizing a nonlinear solver, the vectorized stages are solved as part of the global optimization problem. Because the stage calculation is part of the larger optimization problem, the need to iteratively solve for the state solution at every optimization iteration is eliminated. This leads to a more computationally efficient evaluation of the optimization model as it can be evaluated in a single explicit step. However, the complexity of the optimization problem is increased as there are a significantly larger number of constraints and design variables compared to the Picard iteration and the time-marching approaches. The optimization problem is formulated as

$$\begin{aligned}
\text{minimize} \quad & \mathbf{F}^*, \\
\text{with respect to} \quad & \mathbf{X}^*, \bar{\mathbf{Y}}_{dv}, \bar{\mathbf{y}}_{dv}, \\
\text{subject to} \quad & \mathbf{C}^*, \\
& 0 = \mathbf{C}_1 = \bar{\mathbf{A}}\bar{\mathbf{H}}F(\bar{\mathbf{Y}}_{dv}, \bar{\mathbf{P}}_d, \mathbf{p}_s) + \bar{\mathbf{U}}\bar{\mathbf{y}}_{dv} - \bar{\mathbf{Y}}_{dv}, \\
& 0 = \mathbf{C}_2 = \bar{\mathbf{B}}\bar{\mathbf{H}}F(\bar{\mathbf{Y}}_{dv}, \bar{\mathbf{P}}_d, \mathbf{p}_s) + \bar{\mathbf{V}}\bar{\mathbf{y}}_{dv} - \bar{\mathbf{y}}_{dv},
\end{aligned}$$

where $\mathbf{F}^*$, $\mathbf{X}^*$, and $\mathbf{C}^*$ are the objective, design variables and constraints of the original global optimization problem. Further, $\bar{\mathbf{y}}$ is the vector of states corresponding to the equivalent definition in Equation (31) formed by the initial condition and design variables: $\bar{\mathbf{y}} = [\mathbf{y}_0^T,\ \bar{\mathbf{y}}_{dv}^T]^T$. The constraint $\mathbf{C}_1$ is the formulation of the left side of Equation (32) as an equality constraint where $\bar{\mathbf{Y}}_{dv}$ are the stages $\bar{\mathbf{Y}}$ as design variables. Similarly, $\mathbf{C}_2$ is the equality constraint corresponding to the right side of Equation (32). During an optimization iteration, $\bar{\mathbf{H}}F(\bar{\mathbf{Y}}_{dv}, \bar{\mathbf{P}}_d, \mathbf{p}_s)$ is first computed and used for both $\mathbf{C}_1$ and $\mathbf{C}_2$. Similar to Picard iteration, the coefficient matrices are precomputed and stored using a sparse data structure to decrease computational cost.

The sparsity pattern of the constraint Jacobian is broken up into grids of block matrices as shown in Figure 5. Each state has its own set of constraints, $\mathbf{C}_1$ and $\mathbf{C}_2$, as well as design variables $\bar{\mathbf{Y}}_{dv}$ and $\bar{\mathbf{y}}_{dv}$ to form the outer grid pattern. Furthermore, each state's constraints $\mathbf{C}_2$ is independent of all other states' $\bar{\mathbf{y}}_{dv}$. This is due to the structure of $\bar{\mathbf{V}}$ as well as the fact that $F(\bar{\mathbf{Y}}_{dv}, \bar{\mathbf{P}}_d, \mathbf{p}_s)$ is independent of $\bar{\mathbf{y}}$. Note that at first glance, the block diagonal formulation used thus far from Equation (30) does not seem consistent with this description as $\bar{\mathbf{Y}}_{dv}$ and $\bar{\mathbf{y}}_{dv}$ should

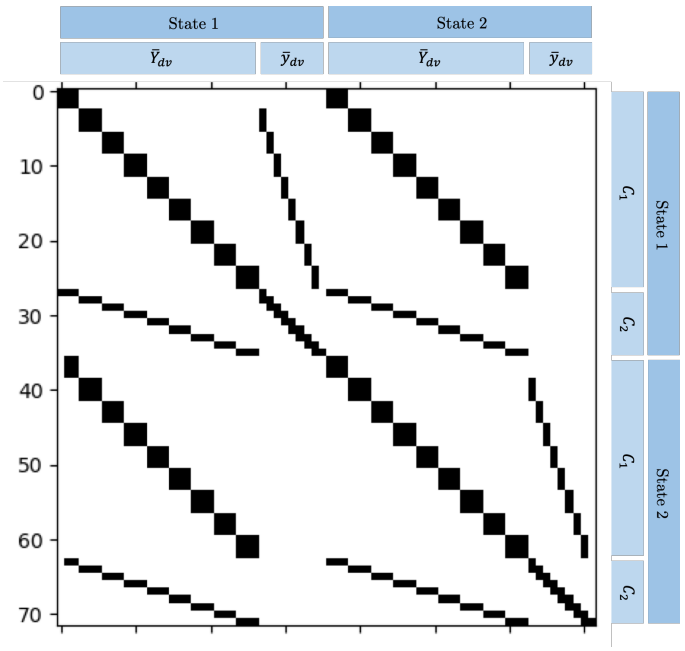

Fig. 5 Example sparsity pattern of the constraint Jacobian for the collocation approach. The sparsity pattern above has $n_y = 2$, $n_t = 10$, and $s = 3$.

consist of all states. However, reordering $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{U}}, \tilde{\mathbf{V}}$ by state instead of by stage will demonstrate that they are equivalent.

Because of the similar formulation to Picard iteration, parallelism across time is possible as the evaluation of $\bar{\mathbf{F}}$ is vectorized. Further, the total derivative computation is also automated by `CSDL` in the same manner. However, because a nonlinear solver is not used within an optimization iteration, the total derivative computation is less computationally expensive than that of Picard iteration.

4 User Interface

This section outlines the workflow for using `Ozone` to solve ordinary differential equations (ODEs) and how `Ozone` interfaces with the Computational System Design Language (`CSDL`). Similar to other ODE solvers, `Ozone` first requires the user to define an ODE function, along with corresponding inputs and outputs. The user must also specify the numerical integration method and solution approach as discussed in the preceding sections. The inputs and outputs are `CSDL Variable` objects that can be passed to and from other parts of the larger computational model, enabling easy integration to other `CSDL` codes.

Listings 1, 2, and 3 show an example of using `Ozone` to solve the Lotka-Volterra equations and computing derivatives through the integrator using `CSDL`'s API.

Since `Ozone` is built upon `CSDL` (Sect. 2.5), users are required to create their optimization model and define their ODE functions using `CSDL`'s mathematical API in Python. `CSDL` offers a syntax similar to NumPy, easing the transition for users familiar with writing numerical code in Python. For more details, users are encouraged to refer to the `CSDL` documentation which is linked in the `Ozone` repository.

4.1 Formulating an ODE Problem

The `ODEProblem` class serves as the core interface for defining and solving ODEs in `Ozone`. To instantiate an `ODEProblem`, the user must specify a numerical integration method (e.g., Runge-Kutta 4th order) and a solution approach (e.g., time-marching). The user must define the ODE

function and specify states, time span and optional dynamic parameters using CSDL `Variables` computed upstream or constant NumPy arrays. The use of `ODEProblem` and its specific syntax can be seen in Listing 1.

Listing 1 Using Ozone's `ODEProblem` to formulate an ODE integration problem. The numerical method, solution approach, states, time span and dynamic parameters are defined using this class. The method and approach can be selected without modifying the rest of the problem formulation.

```
num_times = 10 # number of time points
# Create initial conditions
x_0 = csdl.Variable(name = 'x_0', value = 2.0)
y_0 = csdl.Variable(name = 'y_0', value = 2.0)

# Create dynamic parameter 'a', static parameter 'd'
# dynamic parameter must be defined for every timestep
a = np.zeros((num_times, ))
for t in range(num_times):
    a[t] = 1.0 + t/num_times/5.0
a_dynamic = csdl.Variable(name = 'a', value = a)
# static parameter
d_static = csdl.Variable(name = 'd', value = 0.5)

# Timestep vector with 9 timesteps of size 0.1
h = csdl.Variable(name = 'h', value = np.full(num_times-1, 0.1))

# Choose approach and method
approach = ozone.approaches.TimeMarching()
# approach = ozone.approaches.PicardIteration()
# approach = ozone.approaches.Collocation()

method = ozone.methods.RK4()
# method = ozone.methods.ImplicitMidpoint()

# Initialize ODE problem:
ode_problem = ozone.ODEProblem(method, approach)

# Set inputs/states to your ODE problem.
# Define states with their initial conditions
ode_problem.add_state('y', y_0)
ode_problem.add_state('x', x_0)
# Define dynamic parameters
ode_problem.add_dynamic_parameter('a', a_dynamic)
# Define time span
ode_problem.set_timespan(
    ozone.timespans.StepVector(start=0.0, step_vector=h))
# pass any arguments to ode_function
ode_problem.set_function(ode_function, d_static)
```

When adding dynamic parameters to an `ODEProblem`, the first dimension must have a size of $n_t - 1$, representing the value at each time step. On the other hand, static parameters can have arbitrary shape and are not explicitly declared in `Ozone`. Instead, they can be directly passed in as an argument to the ODE function.

4.2 Defining an ODE Function

The ODE function, which must be given to the `ODEProblem`, is a callable, user-defined Python function representing F that computes time derivatives given states and input parameters. The function's argument is an `ODEVars` object, which provides states and parameters as specified using `ODEProblem`. `ODEVars` is also used to set time derivatives and any outputs. Listing 2 demonstrates an implementation of a typical ODE function, highlighting the use of `ODEVars` to access dynamic parameters and states.

Table 2 n_n is automatically set by `Ozone` for all combinations of methods and approaches.

	Explicit	Implicit
Time-marching	1	s
Time-marching with checkpointing	1	s
Picard iteration	$n_t s$	$n_t s$
Collocation	$n_t s$	$n_t s$

Listing 2 The ODE function for the Lotka-Volterra equations. A field output and profile output (defined in Section 3.1.1).

```
def ode_function(ozone_vars:ozone.ODEVars, d:csdl.Variable):
    a = ozone_vars.dynamic_parameters['a'] # a(t)
    x = ozone_vars.states['x'] # x
    y = ozone_vars.states['y'] # y

    ozone_vars.d_states['y'] = a*y - 0.5*y*x # dy_dt
    ozone_vars.d_states['x'] = 2.0*x*y - d*x # dx_dt

    # any outputs you want to record summed across time
    ozone_vars.field_outputs['field_x'] = x
    # any outputs you want to record across time
    ozone_vars.profile_outputs['x_plus_y'] = x + y
```

An important consideration when implementing the ODE function is that it must be vectorized. The first dimension of all states, dynamic parameters, time derivatives and outputs must have a size of n_n. This vectorization length, n_n, is determined by `Ozone` depending on the method and approach as tabulated on Table 2. If the ODE function is to be run on a parallel computing environment, n_n would correspond to the maximum possible number of ODE functions evaluated in parallel.

4.3 Processing Outputs

Solving the ODE system requires invoking the `solve` method of the ODEProblem class. This method returns `CSDL` variables of all solved field outputs, profile outputs or state histories specified by the user. Because these outputs are `CSDL Variables`, they can be used as inputs to other `CSDL` operations within the broader optimization problem. As an example, the end of Listing 3 shows how to get automatic derivatives of the maximum of a profile output as discussed thus far using `CSDL`'s derivative API.

Listing 3 Invoking the `solve` method integrates the ODE using the method and approach selected by the user. To showcase the benefits of `CSDL`, `CSDL`'s maximum and derivative API is used to compute total derivatives through the integration.

```
# Solve ODE
outputs = ode_problem.solve()

# Get the maximum of the sum of 'x' and 'y' across time using CSDL
x_plus_y = outputs.profile_outputs['x_plus_y']

# Return CSDL variable representing the derivatives of the maximum
# of the sum of 'x' and 'y' across time with respect to
# dynamic parameters, static parameters,
# time steps, and initial conditions
max_x_plus_y = csdl.maximum(x_plus_y)
dsumxy_da = csdl.derivative(max_x_plus_y, a_dynamic)
dsumxy_dd = csdl.derivative(max_x_plus_y, d_static)
dsumxy_dh = csdl.derivative(max_x_plus_y, h)
```

```
dsumxy_dx0 = csdl.derivative(max_x_plus_y, x_0)
```

The information flow relationship between Ozone and CSDL is shown in Figure 6. Ozone utilizes the vectorized ODE function which gets executed whenever an ODE function evaluation is required. This allows for the same implementation of the ODE function to be used for any approach and method, allowing rapid experimentation of different integration configurations.

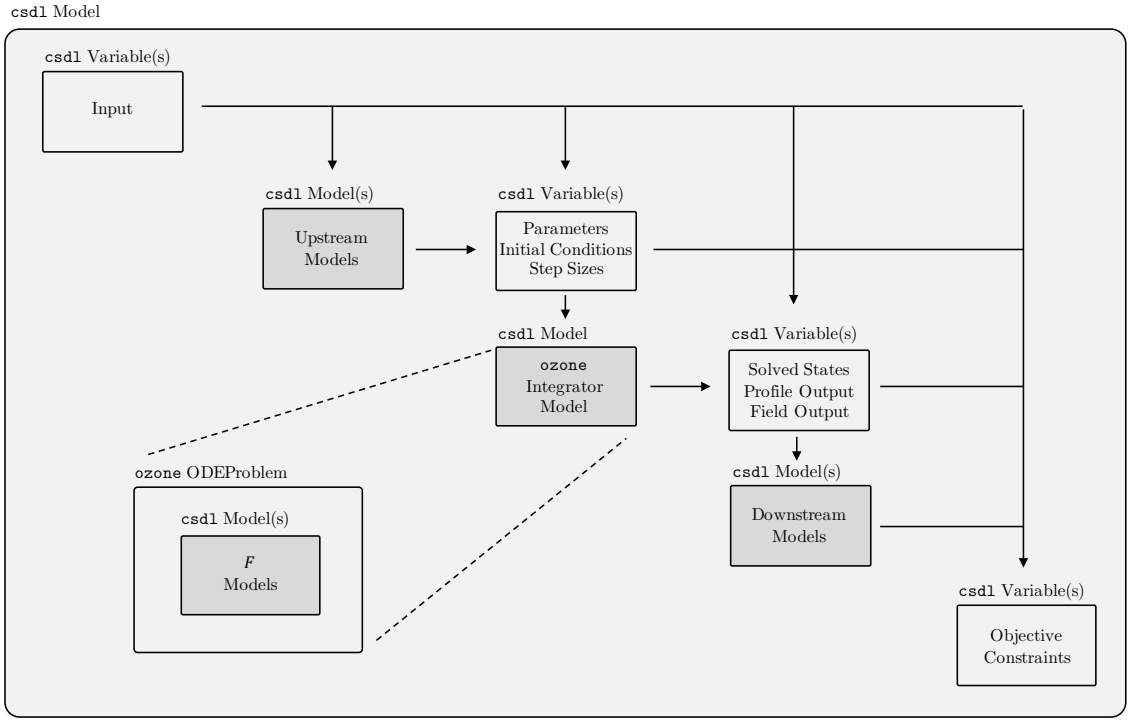

Fig. 6 Flowchart of a CSDL model evaluation containing an ODE integrator from Ozone.

5 Benchmarking Results

In this section, we test the versatility and performance of Ozone by presenting the solution of four real-world optimization problems involving ordinary differential equations (ODEs). We attempt to solve each problem with both low-order and high-order explicit and implicit Runge–Kutta methods using all four solution approaches. Specifically, the tested numerical methods are forward Euler, 4th order Runge–Kutta, implicit midpoint and the 6th order Gauss-Legendre method. Our results highlight strengths and weaknesses of each method and approach with respect to computation time, memory usage and solution accuracy.

We selected a range of real-world examples that have been previously studied to analyze Ozone's capabilities. Each application, also publicly available in the code repository, is implemented using Ozone and CSDL, where assigning the numerical method and the solution approach requires modifications to only a few lines of code as seen in Listing 1. Aside from the approach and method, the optimizations were solved with identical models and parameters. The first of four examples is a lunar ascent trajectory optimization problem from Sandoval et al. (2022) where Picard iteration is the optimal approach. For the next example, we solve an optimal control problem from Dymos (Falck et al., 2021) of a Van der Pol oscillator that shows the advantages of the time-marching approach. The third example is an aircraft trajectory optimization from Orndorff and Hwang (2022) that takes advantage of collocation. Finally, the last example from Armaou and Christofides

Table 3 ODE problem sizes featured in Sect. 5.

Example	n_t	n_y	# of design variables	# of constraints
Lunar Vehicle Ascent	100	7	304	105
Van der Pol Oscillator	40	3	41	0
Trajectory Optimization	39	5	121	5
Nonlinear Reaction-Diffusion Process	40000	210	41	0

(2002) is a nonlinear reaction-diffusion process problem involving a partial differential equation that shows the advantages of time-marching with checkpointing. Table 3 lists the size for each example problem.

To solve each optimization problem, we use the SNOPT (Gill et al., 2005) algorithm interfaced via the modOpt[1] optimization library. The experiments were run on a computer with a 12th generation Intel i7-12700 processor with 64GB of memory using `CSDL`'s JAX backend. To present the computation time for optimization, the convergence history is plotted versus ODE function evaluations. In order to take costs from sensitivity analysis calculations into account, the ODE function derivative computations are included in the evaluation count. Additionally, we provide results where parallelizable ODE function and derivative calls are counted as a single evaluation. Finally, we include plots that highlight the trade-offs between optimization wall time, maximum memory, and relative error for each combination of approach and method. Error is computed relative to the design variable vector obtained from solving the optimization problem with 6th order Runge–Kutta. We omit results for the time-marching with checkpointing approach for all but the last problem as the behavior is similar to time-marching without checkpointing. Methods and approaches that failed to converge or exceeded memory limits are excluded from the plots.

Prior to presenting our results, we highlight some possible sources of inefficiencies of `Ozone` that may cause suboptimal behavior in the performance metrics which will be addressed in the future. As previously mentioned, the parallelized ODE function evaluation has not yet been implemented. Alternatively, vectorization is used, which still provides significant efficiency. Furthermore, although we know the sparsity patterns for the constraint Jacobian in the collocation approach as discussed in Sect. 3.3, the pattern is not explicitly supplied to SNOPT due to limitations in the implementations in `CSDL` and modOpt. However, these limitations can be addressed relatively easily as they only require slight modifications to `CSDL` and modOpt. This may cause suboptimal convergence times compared to the case where the sparsity features are properly utilized. Finally, we note that for all optimization problems, the initial guess of the design variables can have a significant effect on convergence performance. We used initial guesses from the reference sources when possible in order to maintain consistency.

5.1 Lunar Vehicle Ascent

The lunar vehicle ascent problem from Sandoval et al. (2022) minimizes the total time of the ascent of a lunar vehicle. In this optimization, the ODE problem consists of 100 time steps with seven scalar state variables which are defined by

$$\dot{\mathbf{r}} = \mathbf{V}, \qquad \dot{\mathbf{V}} = g(\mathbf{r}) + \frac{T}{m(t)}\mathbf{u}, \qquad \dot{m} = -\frac{T}{v_{ex}}, \tag{35}$$

where $\mathbf{r} \in \mathbb{R}^3$, $\mathbf{V} \in \mathbb{R}^3$ and $m \in \mathbb{R}$ are states and $\mathbf{u} \in \mathbb{R}^3$ is the vector of control inputs. There are three dynamic parameters, which are the control inputs for the x, y and z directions. These time-discretized controls and the final time make up the 304 design variables. The final time is used to control the integration time-step size which in turn controls the ODE integration duration. Further, constraints on the control input u are specified to make up the 101 constraints as well as four additional scalar post-processing constraints. Finally, the objective being minimized is the final time. The optimized trajectory is shown in Figure 7.

[1] modOpt is an open-source optimization library https://github.com/LSDOlab/modopt.

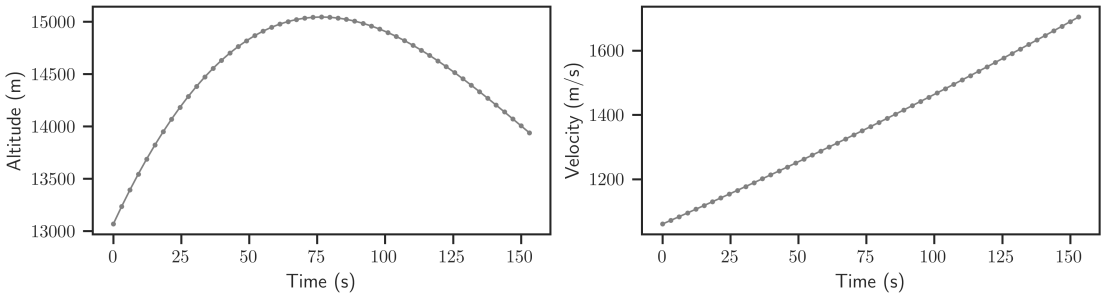

Fig. 7 The optimized trajectory for the lunar vehicle ascent problem.

Figure 8 shows an analysis of the optimization performance for different solution approaches and numerical methods. We can observe that although all approaches and methods lead to a converged optimization solution, their performance characteristics vary. As expected, increasing the order of the numerical method leads to a decrease in relative error as illustrated in the upper middle plot of Figure 8. However, we can also see that the increase in accuracy generally comes with a performance cost due to the increase in the number of stages or the need to solve an implicit system.

In terms of solution approaches, both collocation and Picard iteration require more memory than time-marching due to the need to store large vectors and matrices that scale with n_t. We can also observe relative errors remaining largely unchanged across solution approaches. This is expected as the errors are linked to the numerical method which is independent of how they are solved. Any inconsistencies in the relative error across approaches can be attributed to slightly different optimization solutions for the case of collocation, or differences in nonlinear solver tolerances for Picard iteration.

Figure 8 also highlights large differences in function counts depending on the approach and the level of parallelization. Without any parallelization, the time-marching approach is competitive in terms of function counts. On the other hand, with full parallelization, time-marching is much more expensive as opposed to the other approaches. This is due to the fact that time-marching does not benefit much from parallelization as it is a sequential solver. Further, it is guaranteed that Picard iteration will have more individual function evaluations than time-marching at every time step as the ODE function is evaluated for every time step at least once per optimization iteration. Therefore, without any parallelization or vectorization, time-marching is often the ideal approach.

With full parallelization, Picard iteration and collocation have similar function counts that are far fewer than that of time-marching. The parallelization across the time dimension that is available to both approaches is the main factor contributing to the performance advantage over time-marching. However, the reasons for the performance differences between Picard iteration and collocation are less clear. Picard iteration fully integrates the ODE at every optimization iteration even in cases where the current design variables are far from optimal. On the other hand, collocation does not resolve the ODE solution at every optimization iteration, leading to cheaper forward evaluations. Another possible reason is due to the effect of the nonlinearities of the ODE function on the collocation approach. If the ODE function is linear, the quasi-Newton solver used by the gradient-based optimizer can converge quickly. Hence, the collocation approach may outperform Picard iteration in some cases. It is important to note that the other design variables (the dynamic control inputs in this case) and constraints may negatively affect the convergence even if the ODE is linear. Finally, a good initial guess can give a head start to the collocation approach when the optimization starts. A more detailed study of the convergence properties for Picard iteration and collocation is left for future work.

When considering just optimization wall time, Picard iteration is more performant than both time-marching and collocation for the same error, making it the most effective approach for this problem. This could be due to the reasons outlined above as well as the fact that the implicit function theorem is leveraged to compute derivatives through the nonlinear block Gauss Seidel solver, reducing the number of derivative calculations needed.

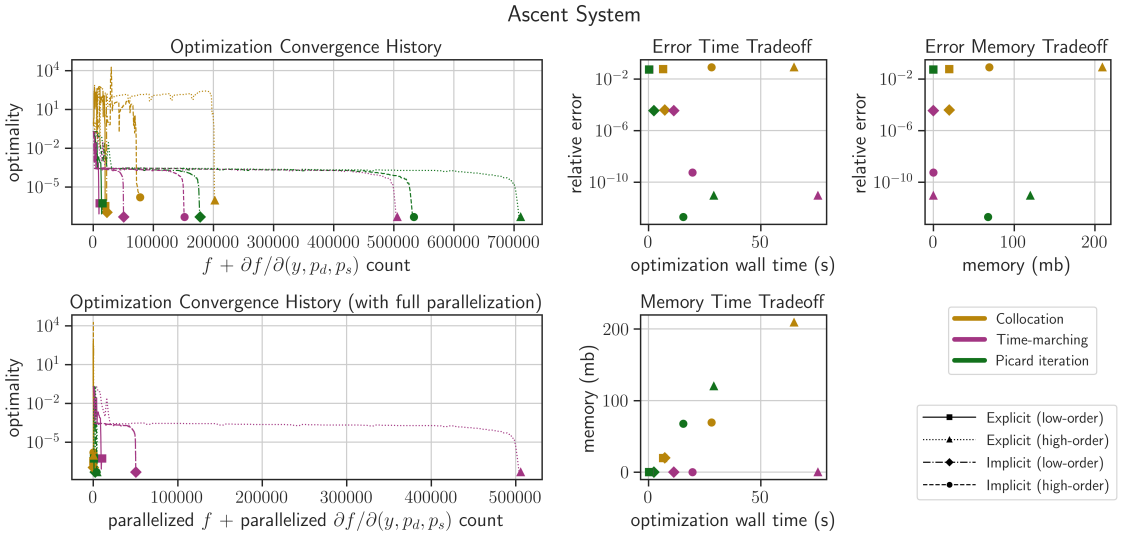

Fig. 8 Performance comparison for the lunar vehicle ascent optimization problem. The colors and line style designate the approach and method (forward Euler, 4th order Runge–Kutta, implicit midpoint, or 6th order Gauss–Legendre) used. All methods and approaches converged.

However, the performance of Picard iteration can suffer if the number of timesteps is too large. Picard iteration involves matrix-vector multiplications with matrices that scale with n_t in both the number of columns and the number of rows. Furthermore, these matrices are not all sparse, e.g the $(\mathcal{I} - \bar{\mathbf{V}})^{-1}$ term in Equation (33) and (34). This results in a time complexity that is not linear with n_t unlike time-marching. Similar behavior also applies for the adjoint-based derivative computation where a dense linear system with rows and columns scaling with n_t must be solved. Further, the ODE function must be suitable for fixed-point iteration, as the nonlinear block Gauss-Seidel solver can diverge. This phenomenon is seen in the following example.

5.2 Van der Pol Oscillator

The Van der Pol oscillator problem from the Dymos (Falck et al., 2021) documentation contains a second-order nonlinear ODE where the goal is to minimize a cost function with respect to control inputs. The Van der Pol ODE with the cost function is described by

$$\dot{x} = (1 - y^2)x - y + u, \qquad \dot{y} = x, \qquad \dot{J} = x^2 + y^2 + u^2,$$

where x and y are two first-order ODEs and J is the cost function. The design variable is a scalar dynamic parameter u which represents the control input. We use 40 time time steps, yielding 41 design variables.

We verify our implementation by comparing our solution with that of Dymos. The left plots of Figure 9 shows the optimized control and the final integrated states for `Ozone` and Dymos using a similar number of collocation points. It can be verified that both implementations converge to approximately the same control and states.

We use this problem as a preliminary benchmark to compare the performance of `Ozone` with other Python ODE solvers, leveraging `CSDL`'s JAX backend. Figure 9 presents the benchmarking results where it can be observed that `Ozone` achieves comparable ODE integration and derivative computation speed relative to other solvers for collocation and time-marching. The solvers we compared against include those that support collocation such as Dymos (Falck et al., 2021) and the preceding `Ozone` implementation in OpenMDAO (Hwang and Munster, 2018). We also compared against the more popular time-marching solvers including Tensorflow's ODE module (Abadi et al., 2015), Scipy (Virtanen et al., 2020), and torchdiffeq for Pytorch (Chen et al., 2018). We attempted to implement the problem efficiently for each solver while maintaining consistency in terms of numerical methods.

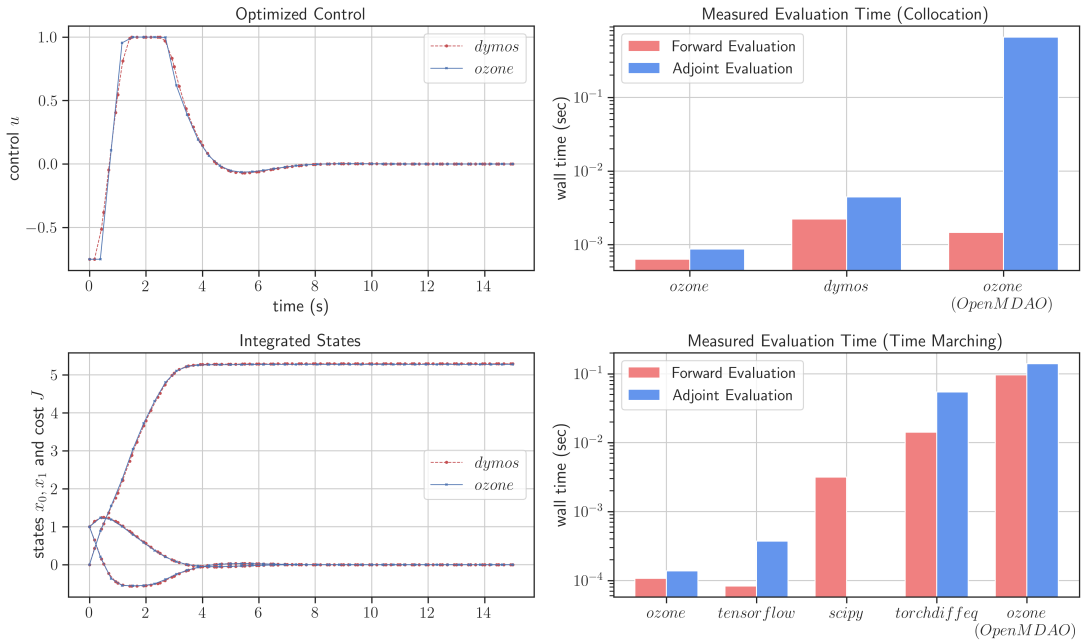

Fig. 9 Verification of the Van der Pol oscillation problem solution and a preliminary, qualitative comparison of the numerical integration and derivative calculation performance. `Ozone` is competitive with other popular Python solvers. Although the preceding `Ozone` implemented in OpenMDAO is the only other solver capable of using both time-marching and collocation, our implementation is significantly faster. We also note that the Scipy ODE solver does not provide derivatives.

We observe several orders-of-magnitude speedups when comparing the current implementation of `Ozone` to its predecessor implemented in OpenMDAO, showing a major benefit to our use of `CSDL` and its JAX backend. The observed performance gains are especially significant against NumPy-based solvers such as SciPy, `Ozone` in OpenMDAO, and Dymos to an extent, while we achieve similar performance to solvers that have a compilation feature such as Tensorflow's ODE module. This aligns with our expectations as `CSDL`'s JAX backend features a Just-in-Time compilation ability that we leverage.

We note that this benchmark should be interpreted as a qualitative comparison to demonstrate Ozone's competitive performance rather than a definitive comparison. Differences in solver implementation of the Van der Pol problem and issues in maintaining full consistency across solvers lead to variations in performance. For example, Tensorflow and Scipy were limited to adaptive time-stepping methods while the others used fixed timestep 4th order Runge-Kutta. Further, we did not take advantage of some advanced features such as Jacobian sparsity that were available in some solvers.

An analysis of the optimization performance between methods and approaches within `Ozone` is shown in Figure 10. Unlike the preceding example, Picard iteration failed to converge for all methods. This is either due to the time discretization being too coarse or the ODE formulation causing instability during the nonlinear Gauss Seidel iteration procedure. It has been shown that there is always a discretization fine enough that will allow Picard iteration to converge (Gandarillas and Hwang, 2019). Picard iteration has been observed to be the least stable approach out of the three. Therefore, it can be recommended to decrease the time step size if Picard iteration is to be used.

Furthermore, it can be noted that the collocation approach converges significantly faster than time-marching in terms of function evaluation count regardless of parallelization level. The reason for this comes down to similar points raised in the previous example where the linearity of the ODE function as well as the initial guess could lead to faster convergence. This demonstrates an advantage of the collocation approach beyond the parallelization aspect.

However, when comparing optimization wall time, the time-marching approach is significantly faster than collocation. Although collocation requires less function evaluations and optimization

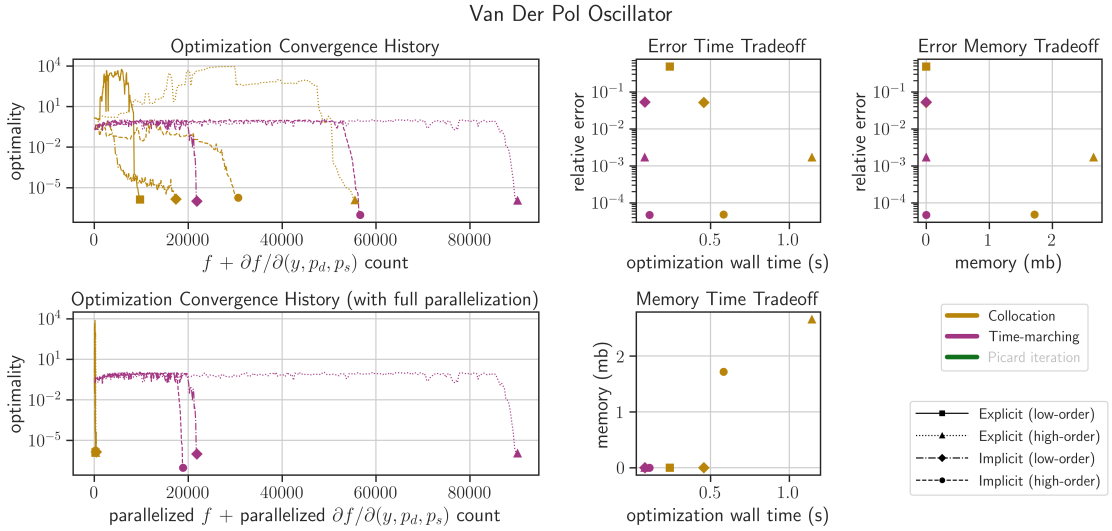

Fig. 10 Performance comparison for the Van Der Pol oscillator optimization problem. The colors and line style designate the approach and method (forward Euler, 4th order Runge–Kutta, implicit midpoint, or 6th order Gauss–Legendre) used. The Picard iteration approach failed to converge due to divergence in the nonlinear Block Gauss–Seidel solver.

iterations, the complexity of the optimization problem is substantially increased due to the addition of multiple design variables and constraints per time step. In short, the collocation approach offloads a large part of the numerical integration task to the optimizer, leading to more time spent within the optimizer itself.

Further, each optimization iteration using the collocation approach requires several large sparse matrix-vector multiplications. As shown in section 3.3, the sparse concatenated GLM matrices $\bar{\mathbf{A}}$, $\bar{\mathbf{B}}$, $\bar{\mathbf{U}}$, and $\bar{\mathbf{V}}$ all scale with n_t, as evidenced by the increased memory usage compared to the time-marching approach.

Finally, we can observe the relative error to be equal between the two approaches for this application. This is expected as the errors from numerical integration are ingrained in the numerical method, not the solution approach. As the time-marching approach offers reduced computation time and memory usage, it is the preferable choice of approach in this scenario. However, for more complex trajectory optimization problems, such as those with complicated controls, dynamics and path constraints, the collocation approach can outperform time-marching. This is explored further in the following section.

5.3 Trajectory Optimization

The trajectory optimization benchmarking problem (Orndorff and Hwang, 2022) is a direct-transcription optimal control problem. A three degree-of-freedom transition maneuver of NASA's lift-plus-cruise concept is optimized considering multiple disciplines. NASA's lift-plus-cruise concept is an air taxi consisting of eight lift rotors and one pusher rotor. The ODE system consists of five states and unlike the preceding examples, the ODE function is more complex due to the inclusion of a surrogate model to predict the axial and tangential velocities of the rotors. For this problem, 39 time steps are used with three dynamic control inputs which consist of 120 design variables. The final design variable is the time step size to control the integration time, similar to the lunar vehicle ascent example in Sect. 5.1. Inequality constraints are added to some minimum and final values of states to serve as path constraints. Further, a post-processing constraint to limit noise using an acoustic model is also included. Finally, the objective being minimized is the total energy used for the maneuver. The optimized trajectory is shown in Figure 11.

A comparison of the two solution approaches that converged can be seen in Figure 12. Similar to the Van der Pol oscillator example, Picard iteration did not converge for any method. However,

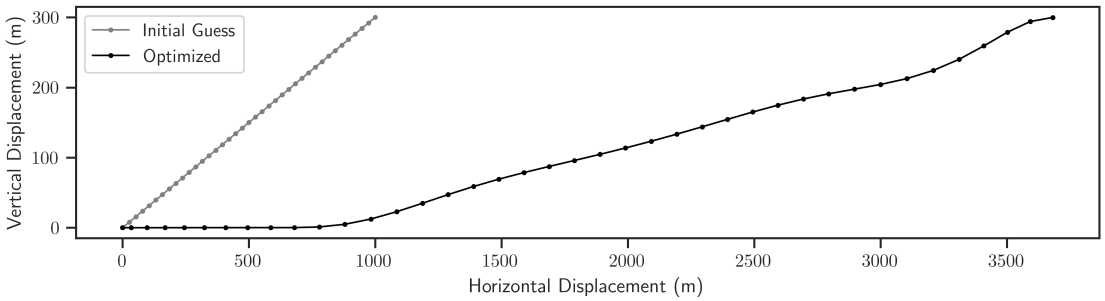

Fig. 11 The minimum energy trajectory for NASA's lift-plus-cruise with final altitude, minimum altitude, final velocity, final flight path angle, and maximum noise constraints.

in contrast to the Van der Pol oscillator problem, the collocation approach converged significantly faster than the time-marching approach both in function count and optimization time.

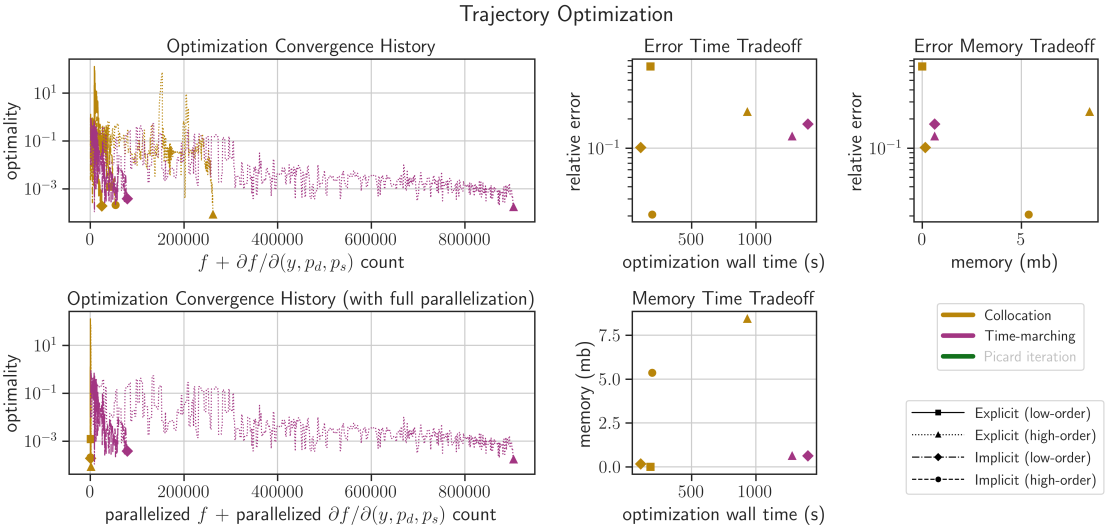

Fig. 12 Performance comparison for the trajectory optimization problem. The colors and line style designate the approach and method (forward Euler, 4th order Runge–Kutta, implicit midpoint, or 6th order Gauss–Legendre) used. The Picard iteration approach failed to converge due to divergence in the nonlinear Block Gauss–Seidel solver.

Time-marching, akin to a single shooting method, can have trouble with trajectory optimization problems with complicated controls and dynamics as well as difficult path constraints. In this problem, the dynamics are fairly complex, involving several surrogate models modeling aerodynamics and propulsion. Further, the imposed minimum height, final height, and maximum noise constraints also add significant complexity to the optimization problem. These factors introduce difficulties to the optimizer using the time-marching approach, which is sensitive to initial conditions.

We also observe only a small performance gap between explicit and implicit methods when using collocation. While implicit methods require solving a nonlinear system every timestep when using time-marching (although it results in smaller error), this is not necessary for collocation. The collocation approach does not explicitly resolve the nonlinear coupling at every optimization iteration. Instead, the optimization constraint Jacobian is a large sparse block matrix, as seen in figure 5, for both implicit and explicit methods, leading to only small differences in performance.

Overall, this problem highlights the advantages of collocation when used for complicated trajectory optimization problems and the flexibility of Ozone, enabling easy switching between solution approaches for analysis

5.4 Nonlinear Reaction-Diffusion Process

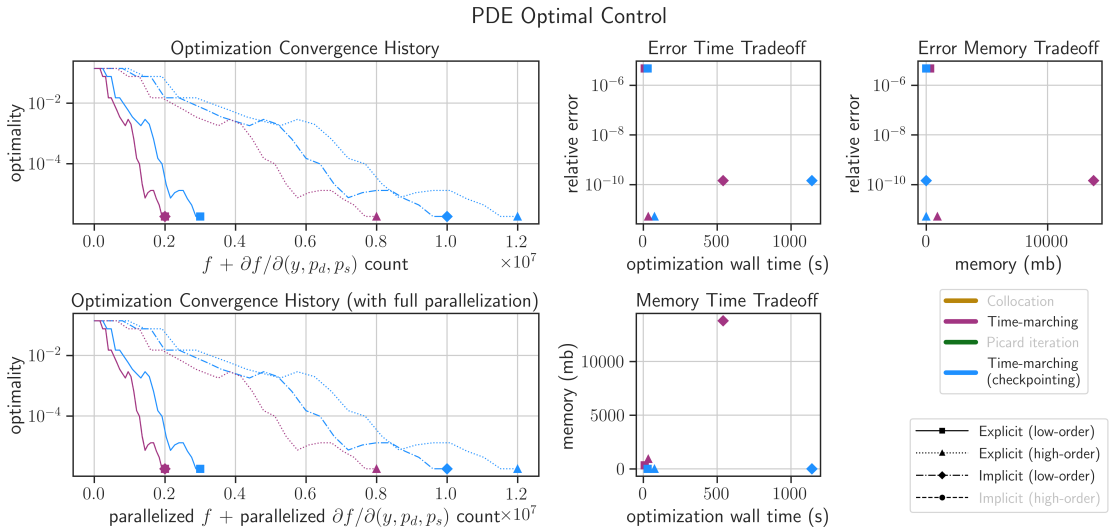

Fig. 13 Performance comparison for the nonlinear reaction-diffusion process optimization problem. The colors and line style designate the approach and method (forward Euler, 4th order Runge–Kutta, implicit midpoint, or 6th order Gauss–Legendre) used. Both collocation and Picard iteration approaches reached memory usage limits. The 6th order Gauss–Legendre method failed to solve the problem due to memory limits without checkpointing and excessive evaluation time with checkpointing.

This section considers a partial differential equation (PDE) modeling a nonlinear diffusion-reaction process on a rod. The PDE in question is from Armaou and Christofides (2002) and is given by

$$\dot{x} = \frac{\partial^2 x}{\partial z^z} + \mathcal{B}_T \left(e^{-\gamma/(1+x)} - e^{-\gamma} \right) + \mathcal{B}_U \left((H(z - 0.3\pi) - H(z - 0.7\pi))u(t) - x \right), \tag{36}$$

where the state x is the temperature along z, which is the distance along the rod, u is a scalar control and H is the Heaviside function. The second spatial derivative of x in the first term of the right hand side is solved using the method of lines with x being discretized into 210 points. Hence, n_y is equal to 210. Further, the time variable was discretized to get $n_t = 40000$ with a final time value of $t = 2$. The objective to minimize, J, is formulated as

$$J = \int_0^2 \int_0^1 100x^2 + 20u^2 dx dt,$$

and is approximated using the profile output function,

$$f_p(x, u) = \sum_{i=0}^{n_y - 1} 100x_i^2 \Delta x + 20u^2,$$

and taking the weighted sum across time. Because a profile output is used, we do not store the full state history at once. The design variable and control profiles are discretized into 41 points and interpolated across all 40000 time steps as a design variable. Finally, the boundary conditions of the ODE are incorporated into the method of lines within the ODE function model. The final time history of the solved PDE is shown in Figure 14.

This problem encountered difficulties when using the Picard iteration and collocation approaches. The Picard iteration method could not handle storage of the dense coefficient matrix $(\mathcal{I} - \bar{\mathbf{V}})^{-1}$. For the collocation approach, compilation of the computational graph failed due to the large values of n_n (at least 40000). Because of difficulties in vectorizing this ODE function efficiently, the `CSDL` compiler is not able to handle the size of computational graph.

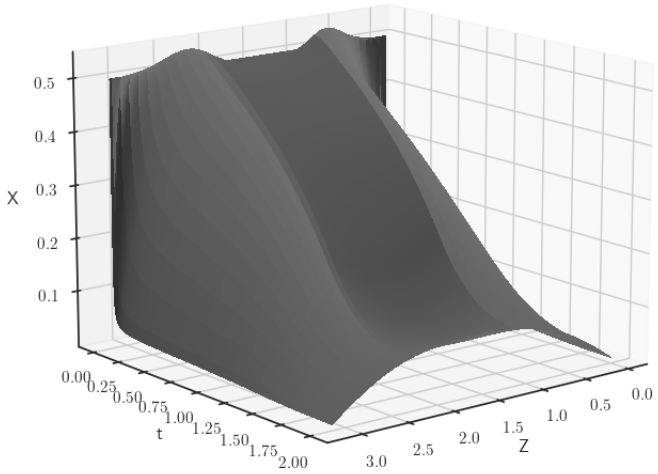

Fig. 14 The time history of the solved state for nonlinear reaction-diffusion process problem.

However, both time-marching with and without checkpointing successfully converged as can be seen in Figure 13. This is because the memory usage associated with them is relatively low compared to Picard iteration and collocation. We highlight the memory discrepancy between the two time-marching approaches. We can see that checkpointing significantly reduces memory consumption by around an order of magnitude as predicted. This is because time-marching without checkpointing stores all intermediate states across all timesteps to improve the adjoint computation time. Thus, as expected, time-marching is faster than time-marching with checkpointing both in measured time and function evaluations. The checkpointing method requires numerical integration of the ODE multiple times during one optimization iteration for the adjoint method, leading to the slower optimization times. This is especially apparent for the implicit methods as linear systems must be solved during the adjoint computation as well.

6 Conclusion

We presented `Ozone`, a general-purpose ordinary differential equation (ODE) solver for gradient-based optimization. `Ozone` enables numerical integration of ODEs using a variety of methods and solution approaches with automatic sensitivity analysis. All methods from the explicit and implicit Runge–Kutta family are available in the implementation. The available solution approaches consist of two serial and two parallel techniques. The serial approaches include time-marching and time-marching with checkpointing, which both sequentially solve for the states through time. Time-marching with checkpointing takes measures to decrease memory costs in exchange for increased computation time. The two parallel approaches include Picard iteration and collocation, which allow vectorized/parallel evaluations of the ODE function. An important feature of `Ozone` is that inputs and outputs to the ODE solver can be connected to other analysis systems easily using `CSDL` and the derivatives of the assembled system can be automatically computed. Another unique advantage of `Ozone` is that it inherits `CSDL`'s ability to execute models using different backends.

The characteristics of the methods and approaches are analyzed through four applications. For each solution approach, we found that there exists at least one case where it exhibits the best performance in terms of time or memory. `Ozone` can therefore reduce the implementation time for ODEs in optimization as it allows seamless switching between the four solution approaches as opposed to using different solvers to find the best approach.

We also found that `Ozone` is proficient in solving a wide range of optimization problems containing ODEs. Our experimental results show that `Ozone` coupled with `CSDL` can find effective solutions for well-known, straightforward optimal control problems without constraints such as the Van der Pol oscillator problem in Sect. 5.2, to complex, PDEs such as the one shown in Sect. 5.4, to problems with multiple disciplines with unconventional functions such as the trajectory optimization problem in Sect. 5.3.

The most promising direction for future work consists in implementing parallelized evaluations of the ODE functions. Because `Ozone` already performs the integration procedure using concatenated values of the state, the required changes will only be on the evaluation of the ODE function. Other potential improvements include implementation of multi-step methods, adaptive timesteps, additional parallel integration approaches, other checkpointing schemes, support for differential algebraic equations and implementing support for passing sparse Jacobian matrices directly to the optimizer. Furthermore, analysis of the example applications demonstrates the need for further study of convergence behavior for Picard iteration and collocation.

Acknowledgements

The authors would like to thank Nicholas C. Orndorff, and Anugrah Jo Joshy for their help in generating the models in Sect. 5.

Authors' contributions

M.S and J.H. conducted the research, performed the analysis, and prepared the main manuscript.

Funding

The material presented in this paper is, in part, based upon work supported by NASA under award No. 80NSSC21M0070.

Availability of data and materials

Not applicable

Declarations

Ethics approval and consent to participate

Not applicable

Consent for publication

Not applicable

Conflict of interest

The authors declare that they have no conflict of interest.

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) TensorFlow: Large-scale machine learning on heterogeneous systems. URL `https://www.tensorflow.org/`, software available from tensorflow.org

Alexe M, Sandu A (2009) Forward and adjoint sensitivity analysis with continuous explicit Runge–Kutta schemes. Applied Mathematics and Computation 208(2):328–346, DOI https://doi.org/10.1016/j.amc.2008.11.035, URL https://www.sciencedirect.com/science/article/pii/S0096300308008576

Armaou A, Christofides PD (2002) Dynamic optimization of dissipative PDE systems using nonlinear order reduction. Chemical Engineering Science 57(24):5083–5114, DOI https://doi.org/10.1016/S0009-2509(02)00419-0, URL https://www.sciencedirect.com/science/article/pii/S0009250902004190

Aupy G, Herrmann J, Hovland P, Robert Y (2016) Optimal multistage algorithm for adjoint computation. SIAM Journal on Scientific Computing 38(3):C232–C255, DOI https://doi.org/10.1137/15M1019222

Avramidis E, Lalik M, Akman OE (2020) Sodecl: An open-source library for calculating multiple orbits of a system of stochastic differential equations in parallel. ACM Trans Math Softw 46(3), DOI https://doi.org/10.1145/3385076

Bai X, Junkins JL (2012) Modified Chebyshev-Picard iteration methods for solution of initial value problems. The Journal of the Astronautical Sciences 59(1):327–351

Bellen A, Zennaro M (1989) Parallel algorithms for initial-value problems for difference and differential equations. Journal of Computational and Applied Mathematics 25(3):341–350, DOI https://doi.org/10.1016/0377-0427(89)90037-X, URL https://www.sciencedirect.com/science/article/pii/037704278990037X

Bellen A, Zennaro M (1993) The use of Runge-Kutta formulae in waveform relaxation methods. Applied Numerical Mathematics 11(1):95–114, DOI https://doi.org/10.1016/0168-9274(93)90042-P, URL https://www.sciencedirect.com/science/article/pii/016892749390042P

Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q (2018) JAX: composable transformations of Python+NumPy programs. URL http://github.com/google/jax

Burrage K (1995) 10. Parallel Methods for Systems of Ordinary Differential Equations, pp 101–120. DOI https://doi.org/10.1137/1.9780898719659.ch10, URL https://epubs.siam.org/doi/abs/10.1137/1.9780898719659.ch10

Burrage K (1997) Parallel methods for ODEs. Advances in Computational Mathematics 7(1):1–31

Butcher JC (2006) General linear methods. Acta Numerica 15:157–256, DOI 10.1017/S0962492906220014

Butcher JC (2016) Numerical methods for ordinary differential equations. John Wiley & Sons

Chen RTQ, Rubanova Y, Bettencourt J, Duvenaud D (2018) Neural ordinary differential equations. Advances in Neural Information Processing Systems

Christianson B (1994) Reverse accumulation and attractive fixed points. Optimization Methods and Software 3(4):311–326

Deshmukh AP, Allison JT (2016) Multidisciplinary dynamic optimization of horizontal axis wind turbine design. Structural and Multidisciplinary Optimization 53(1):15–27

Dutt A, Greengard L, Rokhlin V (2000) Spectral deferred correction methods for ordinary differential equations. BIT Numerical Mathematics 40(2):241–266

Emmett M, Minion M (2012) Toward an efficient parallel in time method for partial differential equations. Communications in Applied Mathematics and Computational Science 7(1):105–132

Eugene W, Moshe Z (1965) On the relation between ordinary and stochastic differential equations. International Journal of Engineering Science 3(2):213–229, DOI https://doi.org/10.1016/0020-7225(65)90045-5, URL https://www.sciencedirect.com/science/article/pii/0020722565900455

Falck R, Gray JS, Ponnapalli K, Wright T (2021) dymos: A Python package for optimal control of multidisciplinary systems. Journal of Open Source Software 6(59):2809, DOI https://doi.org/10.21105/joss.02809

Falgout RD, Manteuffel TA, O'Neill B, Schroder JB (2017) Multigrid reduction in time for nonlinear parabolic problems: A case study. SIAM Journal on Scientific Computing 39(5):S298–S322, DOI https://doi.org/10.1137/16M1082330

Farrell PE, Ham DA, Funke SW, Rognes ME (2012) Automated derivation of the adjoint of high-level transient finite element programs. CoRR abs/1204.5577, URL http://arxiv.org/abs/1204.5577, 1204.5577

Feagin T, Nacozy P (1983) Matrix formulation of the Picard method for parallel computation. Celestial mechanics 29(2):107–115

Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT (2024) A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis. Structural and Multidisciplinary Optimization 67(5):76

Gandarillas VE, Hwang JT (2019) Automatic generation of numerical methods for time integration in large-scale mdo. In: AIAA Aviation 2019 Forum, p 3664

Gander MJ, Vandewalle S (2007) Analysis of the parareal time-parallel time-integration method. SIAM Journal on Scientific Computing 29(2):556–578, DOI https://doi.org/10.1137/05064607X

Gill PE, Murray W, Saunders MA (2005) SNOPT: An SQP algorithm for large-scale constrained optimization. SIAM review 47(1):99–131

Graham KF, Rao AV (2015) Minimum-time trajectory optimization of multiple revolution low-thrust earth-orbit transfers. Journal of Spacecraft and Rockets 52(3):711–727, DOI https://doi.org/10.2514/1.A33187

Gray JS, Hwang JT, Martins JRRA, Moore KT, Naylor BA (2019) OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. Structural and Multidisciplinary Optimization 59(4):1075–1104, DOI https://doi.org/10.1007/s00158-019-02211-z

Griewank A, Walther A (2000) Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Trans Math Softw 26(1):19–45, DOI https://doi.org/10.1145/347837.347846

Griewank A, Walther A (2008) Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM

Hager WW (2000) Runge-Kutta methods in optimal control and the transformed adjoint system. Numerische Mathematik 87:247–282, DOI https://doi.org/10.1007/s002110000178

Hahne J, Friedhoff S, Bolten M (2021) Algorithm 1016: PyMGRIT: A Python package for the parallel-in-time method MGRIT. ACM Trans Math Softw 47(2), DOI https://doi.org/10.1145/3446979

Hwang J, Jasa J, Martins J (2019) High-fidelity design-allocation optimization of a commercial aircraft maximizing airline profit. Journal of Aircraft 56:1–15, DOI https://doi.org/10.2514/1.C035082

Hwang JT, Munster D (2018) Solution of ordinary differential equations in gradient-based multidisciplinary design optimization. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, p 1646

Hwang JT, Lee DY, Cutler JW, Martins JRRA (2014) Large-scale multidisciplinary optimization of a small satellite's design and operation. Journal of Spacecraft and Rockets 51(5):1648–1663, DOI https://doi.org/10.2514/1.A32751

Janssen J, Vandewalle S (1997) On SOR waveform relaxation methods. SIAM Journal on Numerical Analysis 34(6):2456–2481, DOI https://doi.org/10.1137/S0036142995294292

Lions J (1971) Optimal control of systems governed by partial differential equations

Martins J, Hwang J (2013) Review and unification of methods for computing derivatives of multidisciplinary computational models. AIAA Journal 51:2582–2599, DOI https://doi.org/10.2514/1.J052184

Nadarajah S, Jameson A (2000) A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. In: 38th Aerospace Sciences Meeting and Exhibit, p 667

Nolasco E, Vassiliadis VS, Kähm W, Adloor SD, Ismaili RA, Conejeros R, Espaas T, Gangadharan N, Mappas V, Scott F, Zhang Q (2021) Optimal control in chemical engineering: Past, present and future. Computers & Chemical Engineering 155:107528, DOI https://doi.org/10.1016/j.compchemeng.2021.107528, URL `https://www.sciencedirect.com/science/article/pii/S0098135421003069`

Nørsett SP, Simonsen HH (1989) Aspects of parallel Runge-Kutta methods. In: Numerical methods for ordinary differential equations, Springer, pp 103–117

Orndorff NC, Hwang JT (2022) Investigation of optimal air-taxi transition profiles using direct-transcription trajectory optimization. In: AIAA AVIATION 2022 Forum, DOI https://doi.org/10.2514/6.2022-3485, URL `https://arc.aiaa.org/doi/abs/10.2514/6.2022-3485`

Orndorff NC, Sarojini D, Scotzniovsky L, Gill H, Lee S, Cheng Z, Zhao S, Mi C, Hwang JT (2023) Air-taxi transition trajectory optimization with physics-based models. DOI https://doi.org/10.

2514/6.2023-0324, URL https://arc.aiaa.org/doi/abs/10.2514/6.2023-0324
Patterson M, Rao A (2014) GPOPS-II: A MATLAB software for solving multiple-phase optimal control problems using hp-adaptive Gaussian quadrature collocation methods and sparse nonlinear programming. ACM Transactions on Mathematical Software 41:1–37, DOI https://doi.org/10.1145/2558904
Petzold L (1982) Differential/algebraic equations are not ODE's. SIAM Journal on Scientific and Statistical Computing 3(3):367–384, DOI https://doi.org/10.1137/0903023
Rackauckas C, Nie Q (2017) Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia. Journal of open research software 5(1)
Read J, Bani Younes A, Junkins J (2016) Efficient orbit propagation of orbital elements using modified Chebyshev Picard iteration method. CMES - Computer Modeling in Engineering and Sciences 111:65–81
Sadiku M, Obiozor C (2000) A simple introduction to the method of lines. International journal of electrical engineering education 37(3):282–296
Sand J, Burrage K (1998) A Jacobi waveform relaxation method for ODEs. SIAM Journal on Scientific Computing 20(2):534–552, DOI https://doi.org/10.1137/S1064827596306562
Sandoval SA, Lu P, Hwang JT, Rea JR, Sostaric RR (2022) Multiple optima in abort ascent during lunar powered descent. In: AIAA SCITECH 2022 Forum, DOI https://doi.org/10.2514/6.2022-0949, URL https://arc.aiaa.org/doi/abs/10.2514/6.2022-0949
Sandoval SA, Lu P, Hwang JT (2024) Abort guidance during lunar powered descent. Journal of Guidance, Control, and Dynamics 0(0):1–13, DOI https://doi.org/10.2514/1.G007086
Serban R, Hindmarsh AC (2003) CVODES: An ODE solver with sensitivity analysis capabilities. Tech. rep., Technical Report UCRL-JP-200039, Lawrence Livermore National Laboratory
Skeel RD, Tam HW (1992) Limits of parallelism in explicit ODE methods. Numerical Algorithms 2(3):337–349
Sperry M, Kondap K, Hwang JT (2023) Automatic adjoint sensitivity analysis of models for large-scale multidisciplinary design optimization. In: AIAA AVIATION 2023 Forum, p 3721
Tsang T, Himmelblau D, Edgar TF (1975) Optimal control via collocation and non-linear programming. International Journal of Control 21(5):763–768
Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, Polat I, Feng Y, Moore EW, VanderPlas J, Laxalde D, Perktold J, Cimrman R, Henriksen I, Quintero EA, Harris CR, Archibald AM, Ribeiro AH, Pedregosa F, van Mulbregt P, SciPy 10 Contributors (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods 17:261–272, DOI https://doi.org/10.1038/s41592-019-0686-2
Wang Q, Moin P, Iaccarino G (2009) Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. SIAM Journal on Scientific Computing 31(4):2549–2567
Wang Y, Ni G, Liu Y (2020) Multistep Newton–Picard method for nonlinear differential equations. Journal of Guidance, Control, and Dynamics 43(11):2148–2155, DOI https://doi.org/10.2514/1.G005124
Yan J, Li N, Luo T, Tolley MT, Hwang JT (2019) Optimal control and design of an underactuated ball-pitching robotic arm using large-scale multidisciplinary optimization. In: AIAA Aviation 2019 Forum, DOI https://doi.org/10.2514/6.2019-3450, URL https://arc.aiaa.org/doi/abs/10.2514/6.2019-3450
Yan J, Sperry M, Kamensky D, Hwang JT (2021) Dynamic topology optimization of battery packs for eVTOL aircraft under time-dependent loading. DOI https://doi.org/10.2514/6.2021-3068, URL https://arc.aiaa.org/doi/abs/10.2514/6.2021-3068
Zhang H, Sandu A (2014) FATODE: A library for forward, adjoint, and tangent linear integration of odes. SIAM Journal on Scientific Computing 36(5):C504–C523, DOI https://doi.org/10.1137/130912335
Zhang H, Constantinescu EM, Smith BF (2019) PETSc TSAdjoint: a discrete adjoint ODE solver for first-order and second-order sensitivity analysis. CoRR abs/1912.07696, URL http://arxiv.org/abs/1912.07696, 1912.07696