Solution of ordinary differential equations in gradient-based multidisciplinary design optimization

John T. Hwang [*]
NASA Glenn Research Center, 21000 Brookpark Rd, Cleveland, OH 44135
Peerless Technologies Corporation, 2300 National Rd, Beavercreek, OH 45324

Drayton W. Munster [†]
NASA Glenn Research Center, 21000 Brookpark Rd, Cleveland, OH 44135

A gradient-based approach to multidisciplinary design optimization enables efficient scalability to large numbers of design variables. However, the need for derivatives causes difficulties when integrating ordinary differential equations in models. To simplify this, we propose the use of the general linear methods framework, which unifies all Runge–Kutta and linear multistep methods. This approach enables rapid implementation of integration methods without the need to differentiate each one, even in a gradient-based optimization context. We also develop a new parallel time integration algorithm that enables vectorization across time steps. We present a set of benchmarking results using a stiff ODE, a non-stiff nonlinear ODE, and an orbital dynamics ODE, and compare integration methods. In a modular gradient-based multidisciplinary design optimization context, we find that the new parallel time integration algorithm with high-order implicit methods, especially Gauss–Legendre collocation, is the best choice for a broad range of problems.

I. Introduction

Ordinary differential equations (ODEs) arise frequently in computational models for engineering design. In many cases, they define an initial value problem (IVP) where some quantity is used over the course of time. Examples in the aerospace field include an ODE that characterizes the state of charge of a battery or the fuel weight of an aircraft, both of which decrease at some rate over the course of the mission. ODEs also arise in trajectory optimization and in time-dependent structural or aeroelastic problems as a result of Newton's second law.

In this paper, we are primarily motivated by ODEs involved in multidisciplinary design optimization (MDO) problems. In particular, we are interested in large-scale MDO applications, where there is a large number of design variables, and thus, a gradient-based approach is necessary to keep computational costs manageable. The adjoint method provides significant computational speed-up in derivative computation when the model contains any coupling—either within one or more components via systems of equations or between components via feedback loops. The gradient computation time for the adjoint method is effectively independent of the number of design variables; however, its main drawback is that it is complex and time-consuming to implement and to maintain in response to structural changes to the model.

The implementation of the adjoint method is simplified by the modular analysis and unified derivatives (MAUD) approach [1]. It was shown that the adjoint method can be generalized and unified with all other methods for computing derivatives so that regardless of the model structure, a single matrix equation need be solved [2]. Using this unifying matrix equation, a software framework can be built that automates part of the effort required for computing gradients for MDO. In the MAUD approach, the model is built in a modular manner, decomposed into components. Given the partial derivatives of each component's outputs with respect to its inputs, the framework solves the unifying equation to automatically apply the right method (e.g., chain rule, adjoint method) to compute the model-level total derivatives. Therefore, when developing a model that incorporates one or more ODEs, using MAUD simplifies the computation of the derivatives of the objective and constraint functions with respect to the design variables.

There are three aspects that provide the motivation for this paper. First, there is the need for partial derivatives in the components that perform the ODE integration. We require the derivatives of the stages of the integration method if we use a multi-stage solver (e.g., 4th order Runge–Kutta), and those of the starting method if we use a multi-step solver. Although this is a matter of implementation, we present in this paper a general approach that eliminates the

[*]Research engineer (contractor at NASA GRC), AIAA Member.
[†]NASA Pathways Intern, Information and Applications Branch, drayton.w.munster@nasa.gov.

effort required for differentiating the stages and steps when implementing any new Runge–Kutta or linear multistep method. Second, since we are solving an MDO problem, it is an open question how to integrate the ODE. We can take a conventional time-marching approach, but two other options are treating the ODE state variables as design variables in the optimization problem (as is typically done in direct-transcription optimal control) and treating them as unknowns in a nonlinear system (i.e., multiple shooting methods). We compare the performance of these three approaches within the MAUD context. Third, we also provide some guidance on which ODE integration methods are appropriate for solving typical ODEs by presenting benchmarking results for ODEs of various types.

We implement all methods and generate all results using OpenMDAO, a NASA-funded open-source software framework in Python, developed primarily to facilitate gradient-based MDO [3]. OpenMDAO uses the MAUD approach; therefore, it provides a platform that is sufficient for achieving the goals of this paper. The ODE solver library that was developed as part of this work is expected to become open-source in the near future, and thus it will be available to use for OpenMDAO users.

II. Background

Before presenting the new methods, we review the prior work in terms of ODE formulations, optimal control, and integration methods.

A. ODE formulations

The general form of an ODE is given by

$$y' = f(t, y), \tag{1}$$

where y is the ODE state variable vector, f is the ODE function, and t represents time but can be any independent variable in general. The typical approach for solving an ODE is time-marching. If the method is explicit, the formulas can be evaluated in sequence with no iterative solution required at any point. If the method is implicit, a system of equations must be solved at every time step.

An alternative class of approaches is parallel-in-time integration, which has been studied for over 5 decades [4]. Parallel time integration is typically motivated by the solution of partial differential equations (PDEs), where f is expensive to evaluate and y is a large vector. The application of the adjoint method in a time-dependent simulation provides another motivation for a parallel approach because the full time history of all states is required. The resulting memory requirements are alleviated by parallelizing in time when the parallelization in space is saturated.

In a review of parallel time integration methods, Gander [5] describes 4 types of methods: shooting methods, domain decomposition, multigrid, and direct solvers. However, we only discuss shooting methods in detail as the other three are specific to PDEs. Nievergelt developed the first parallel time integration method that divides the time interval into sub-intervals that are integrated in parallel [4]. Bellen and Zennaro [6] use a multiple shooting method where the state variable at each time step is an unknown in a nonlinear system solved using a variant of Newton's method. Chartier and Philippe [7] also use a multiple shooting method, but here the time interval is split into subintervals and only the final state values of those sub-intervals are unknowns solved using Newton's method. Lions et al. developed the parareal method [8], which uses sub-intervals as well, but instead of using Newton's method, time integration is iteratively performed on the coarse mesh comprised of the endpoints of the sub-intervals. The results of the coarse integration are used to update the initial conditions of the fine integration in each iteration.

Parallel time integration methods suffer from computational redundancy since they do not take advantage of the natural flow of information in the forward time direction. Previously, the motivation for this was due to the computational costs, memory costs, or both being prohibitive when limited to parallelizing in space. In the context of this paper, we have another reason to parallelize in time. Since we use the MAUD approach in the OpenMDAO software framework, time-marching is implemented by instantiating separate instances of the components for the ODE integration method and the ODE function at each time step. However, this incurs significant overhead, especially due to the framework-level data transfers between each time step. This cost is especially noticeable when f is fast to evaluate and there is a large number of time steps. In these cases, it is actually faster to use a parallel integration approach despite the redundant computations because the evaluation of f can be vectorized across time steps, unlike in the time-marching situation.

B. Optimal control

Optimal control problems represent the system dynamics using an ODE. The control variables in optimal control are design variables in MDO, and there are two ways of formulating these. In indirect transcription, the optimality conditions are hand-derived and specialized techniques are used to compute the solutions of the optimality conditions and ODE simultaneously. In direct transcription, the design variables are controlled by the optimizer. Therefore, unlike indirect transcription, the direct transcription approach can be implemented using MAUD and OpenMDAO, and the methods presented in this paper apply.

C. ODE integration methods

In this section, we review the Runge–Kutta and linear multistep methods. A significant portion of the commonly used ODE integration methods fall under the umbrella of these two categories, including pseudospectral methods and predictor-corrector methods, among others. In the equations that follow, h represents the step size, and y_k represents the state vector at the kth time step.

LINEAR MULTISTEP METHODS Linear multistep methods depend on the state and derivative values of not just the previous time step, but also on those of one or more time steps before the previous time step. The advantage of multistep methods is that they can have higher order without requiring additional derivative evaluations because they reuse previous information; however, they sacrifice stability, especially in the explicit forms. They are given by

$$\begin{aligned} y_n &= \alpha_1 y_{n-1} + \cdots + \alpha_k y_{n-k} \\ &\quad + h\beta_0 f(t_n, y_n) + h\beta_1 f(t_{n-1}, y_{n-1}) + \cdots + h\beta_k f(t_{n-k}, y_{n-k}), \end{aligned} \tag{2}$$

where $\alpha_1, \ldots, \alpha_k$ and $\beta_0, \ldots, \beta_k$ are scalar coefficients.

The original linear multistep methods were the Adams-Bashforth methods, which are explicit methods in which the current state value depends on derivatives from more than one previous time step. For the Adams-Bashforth methods, $\alpha_2, \ldots, \alpha_k$ are zero, $\alpha_1 = 1$ and $\beta_0 = 0$. The Adams-Moulton methods are similar, but are implicit rather than explicit. Therefore, $\alpha_2, \ldots, \alpha_k$ are zero and $\alpha_1 = 1$, but β_0 is not zero. The backwards differentiation formulas (BDF) are also implicit methods; however, they are multistep in that the formula depends on state values, rather than derivatives from more than one time step. Therefore, $\beta_1, \ldots, \beta_k$ are zero and $\beta_0 = 1$. For all multistep methods, there are conditions on the coefficients that must be satisfied for consistency. Predictor-corrector Adams methods use an Adams-Bashforth formula to predict y_n and then use this value to evaluate an Adams-Moulton formula without having to solving a system of equations, although we note that there are many variations on this basic concept.

RUNGE–KUTTA METHODS Unlike linear multistep methods, Runge–Kutta methods have a single step, but multiple stages. Assuming s is the number of stages, they are given by

$$Y_1 = y_{n-1} + ha_{11} f(t, Y_1) + \cdots + ha_{1s} f(t, Y_s) \tag{3}$$

$$\vdots \tag{4}$$

$$Y_s = y_{n-1} + ha_{s1} f(t, Y_1) + \cdots + ha_{ss} f(t, Y_s) \tag{5}$$

$$y_n = y_{n-1} + hb_1 f(t, Y_1) + \cdots + hb_s f(t, Y_s), \tag{6}$$

where $Y_1, \ldots, Y_s$ are the state values for the s stages. In a similar manner to the linear multistep methods, there are conditions on the a_{**} and b_* coefficients for consistency. When the $[a_{**}]$ matrix is strictly lower triangular, it corresponds to an explicit Runge–Kutta method. For the same number of time steps, explicit Runge–Kutta methods require more evaluations than explicit linear multistep methods; however, they have better stability properties.

COLLOCATION METHODS Collocation methods used in optimal control (see, for example, Topputo and Zhang [9]) fall under the category of implicit Runge–Kutta methods as they can also be expressed in this mathematical form. Collocation methods discretize each element (i.e., time step) into nodes, and at a subset of these nodes, the interpolated and computed derivative are constrained to be equal. The high-order Gauss–Lobatto methods use Hermite interpolation while the pseudospectral method uses Lagrange interpolation to compute the interpolated derivatives. In the end, the collocation constraints can be expressed as linear functions of the states and computed derivatives at the nodes—these can be interpreted as stage values and stage derivatives, respectively, if viewed as Runge–Kutta methods. Through inversion of matrices from these linear functions, collocation methods can be expressed in the general Runge–Kutta form given above.

III. Methods

We now describe a mathematical framework called general linear methods (GLMs) that unifies all Runge–Kutta and linear multistep methods. We show how implementing this generalized set of equations and its partial derivatives allows any Runge–Kutta and linear multistep method to be easily used without the need to differentiate it. We also present 3 ways of solving the ODEs: time-marching, solver-based, and optimizer-based.

A. General linear methods

The general linear methods (GLM) [10] refer to a generalization of all linear multistep and Runge–Kutta methods. The unified mathematical form allows for multistage and multistep methods that are implicit or explicit. The generalization also provides a simple mathematical framework for incorporating and analyzing more recent methods that do not fit into the traditional linear multistep or Runge–Kutta categories.

Our notation for GLMs assumes r steps and s stages. As with the Runge–Kutta methods, let $Y_1, \dots, Y_s$ denote the state values for the s stages. Following Burrage and Butcher [11], we define $Y, F \in \mathbb{R}^{sN}$ and $y^{[n-1]}, y^{[n]} \in \mathbb{R}^{rN}$ where

$$Y = \begin{bmatrix} Y_1 \\ \vdots \\ Y_s \end{bmatrix}, \quad F = \begin{bmatrix} F_1 \\ \vdots \\ F_s \end{bmatrix}, \quad y^{[n-1]} = \begin{bmatrix} y_1^{[n-1]} \\ \vdots \\ y_r^{[n-1]} \end{bmatrix}, \quad y^{[n]} = \begin{bmatrix} y_1^{[n]} \\ \vdots \\ y_r^{[n]} \end{bmatrix}, \tag{7}$$

where $Y_1, \dots, Y_s$ are the states at the s stages; $F_1, \dots, F_s$ are the derivatives at the s stages; $y_1^{[n-1]}, \dots, y_r^{[n-1]}$ are the states from the steps available at the previous time step; and $y_1^{[n]}, \dots, y_r^{[n]}$ are the states from the steps available at the current time step.

The GLM equations are given by

$$\begin{aligned} Y_i &= \sum_{j=1}^{s} h a_{ij} F_j + \sum_{j=1}^{r} u_{ij} y_j^{[n-1]}, \quad i = 1, \dots, s, \\ y_i^{[n]} &= \sum_{j=1}^{s} h b_{ij} F_j + \sum_{j=1}^{r} v_{ij} y_j^{[n-1]}, \quad i = 1, \dots, r, \end{aligned} \tag{8}$$

where the first equation computes the stages and the second computes the new state values at the current time step. Representing the $a_{**}, b_{**}, u_{**}, v_{**}$ coefficients using matrices A, B, U, V, the GLM equations can be compactly written as

$$\begin{bmatrix} Y \\ y^{[n]} \end{bmatrix} = \begin{bmatrix} A \otimes \mathcal{I} & U \otimes \mathcal{I} \\ B \otimes \mathcal{I} & V \otimes \mathcal{I} \end{bmatrix} \begin{bmatrix} hF \\ y^{[n-1]} \end{bmatrix}, \tag{9}$$

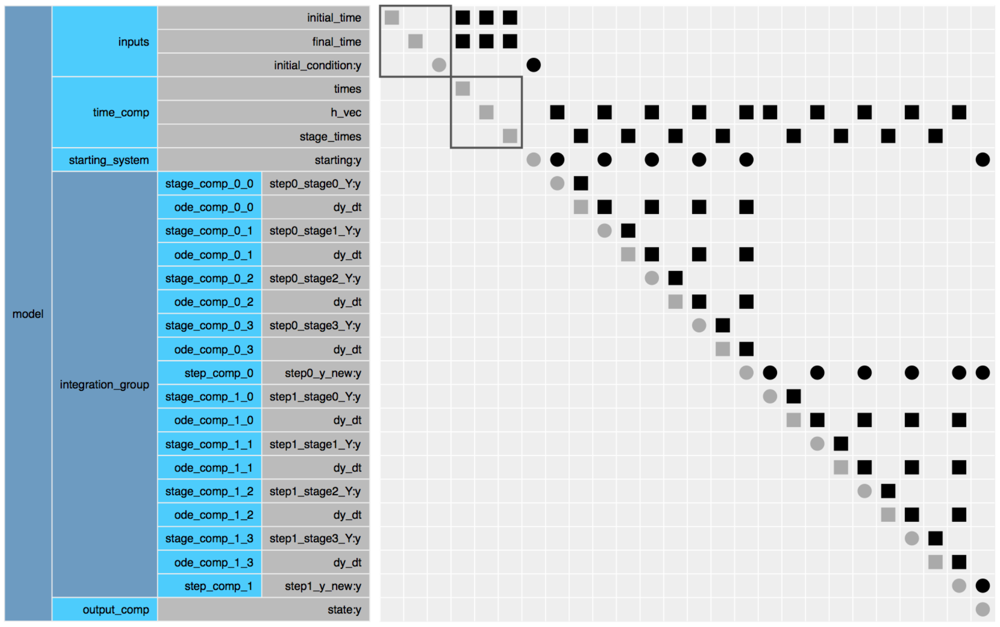
where $\mathcal{I}$ is the identity matrix.

The significance of this formulation is that all linear multistep and Runge–Kutta methods can be expressed in this form with the right choice of the matrices A, B, U, V. In the MAUD approach, the GLM equations and its partial derivatives can be implemented once for all Runge–Kutta and linear multistep methods. Any new method can be implemented simply by defining the correct A, B, U, V matrices.

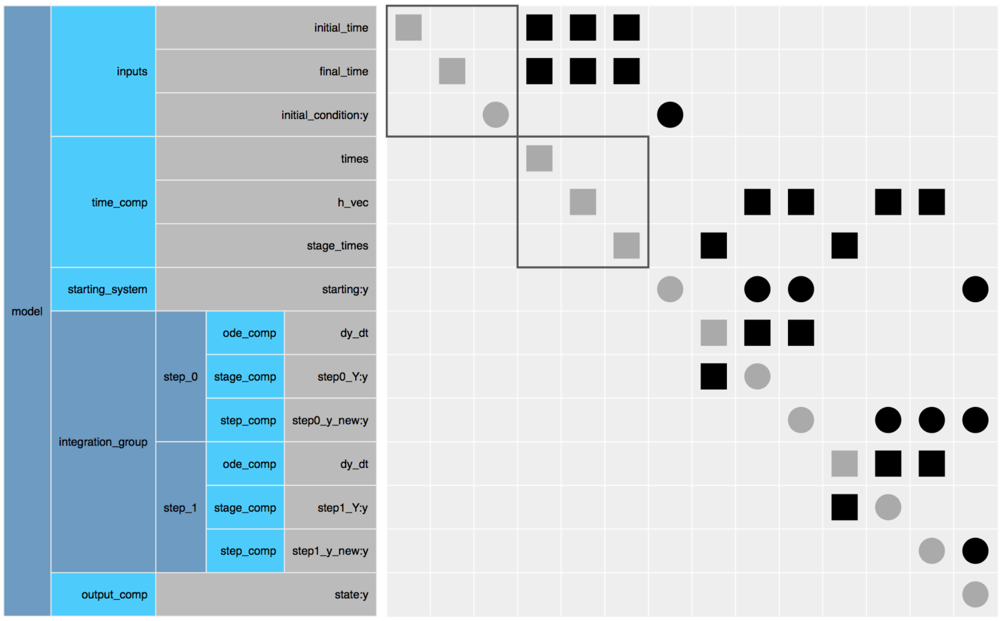
B. Solution approaches

Here, we describe 3 solution approaches that apply regardless of the integration method: time-marching, solver-based, and optimizer-based. For each, we show in Figs. 1, 2, and 3 the design structure matrix for two time steps of an explicit method, the 4th order Runge–Kutta method (RK4), and an implicit method, the implicit midpoint method (IM). These diagrams show the hierarchy of the model as implemented in OpenMDAO as well as the dependency graph. The darker blue shows OpenMDAO *Group* objects that contain components and other groups, and the lighter blue show OpenMDAO *Component* objects. The variables are shown in grey. Black entries in the off-diagonals of the dependency graph show data flow where an entry on the upper-triangular portion indicates data flowing towards the bottom right and one on the lower-triangular portion indicates data flowing towards the upper left.

1. TIME-MARCHING APPROACH In the time-marching approach, we evaluate the time steps in sequence. Therefore, we instantiate separate components for each time step and each stage within each time step, as shown in Fig. 1. For RK4, we can see the 2 time steps and the 4 stage components within each time step. As we can see from Eq. (8), we evaluate the stage equation and the ODE function, alternating between the two, and then we evaluate the step equation at the end of the time step. For IM, we can see the feedback loop between the stage equation and the ODE function, necessitating the use of a solver within each time step, as is conventional in implicit time integration.



An explicit method: 4th order Runge–Kutta (RK4)



An implicit method: implicit midpoint (IM)

Figure 1: Design structure matrices for the time-marching approach.

2. Solver-based approach In the solver-based approach, we treat the ODE state variables across all time steps as the unknowns of a nonlinear system. In Fig. 2, we see the same coupled structure for RK4 as we see for IM. For explicit methods, the state variables naturally have a sequential structure with no feedback loops, and for implicit methods, the coupling is restricted to within each time step. However, the solver-based approach results in a coupled structure for all methods, which results in redundant computation at later time steps. The benefit is that this approach enables vectorization of f across time steps, resulting in fewer components and less overhead.

The nonlinear system contains a component for the ODE function and one for the stage and step evaluation. The approach we are presenting is to use a nonlinear block Gauss–Seidel (NBGS) algorithm for solving this system, as Newton's method is slow due to the assembly of the Jacobian matrix and the solution of the resulting linear system. The NBGS approach can be interpreted as alternating between evaluating f given the most recent guess for the stage values and then using the known stage derivatives to compute the new stage and step values according to Eq. (8).

Let us rewrite Eq. (8) in matrix form with all time steps included. We have

$$\begin{bmatrix} \bar{Y} \\ \bar{y} \end{bmatrix} = \begin{bmatrix} \bar{A} & \bar{U} \\ \bar{B} & \bar{V} \end{bmatrix} \begin{bmatrix} \bar{H}\bar{F} \\ \bar{y} \end{bmatrix}, \tag{10}$$

where $\bar{Y}$ is a vector of stage values from all the time steps, $\bar{y}$ is a vector of step values from all the time steps, $\bar{F}$ is a vector of stage derivatives from all the time steps, $\bar{H}$ is a diagonal matrix of step sizes, and $\bar{A}, \bar{B}, \bar{U}, \bar{V}$ are the matrices obtained by expanding A, B, U, V to all time steps. The solution of this system for $\bar{Y}$ and $\bar{y}$ corresponds to the 'vectorized_stagestep_comp' component in Fig. 2.

We now rearrange to form a linear system in $\bar{Y}$ and $\bar{y}$ with $\bar{F}$ assumed known and fixed. We obtain

$$\begin{bmatrix} \mathcal{I} & -\bar{U} \\ 0 & \mathcal{I} - \bar{V} \end{bmatrix} \begin{bmatrix} \bar{Y} \\ \bar{y} \end{bmatrix} = \begin{bmatrix} \bar{A}\bar{H}\bar{F} \\ \bar{B}\bar{H}\bar{F} \end{bmatrix}. \tag{11}$$

We note that the matrix of this linear system depends only on U, V, which do not change during the iterative solution of the ODE nor during the larger optimization loop. Therefore, we can factorize it and back-substitute very cheaply each time we want to solve Eq. (10).

To summarize the solver-based approach, we can efficiently compute the solution of the ODE in a vectorized manner by alternating between evaluating

$$\bar{F} = f(\bar{Y}) \tag{12}$$

and

$$\begin{bmatrix} \bar{Y} \\ \bar{y} \end{bmatrix} = \begin{bmatrix} \mathcal{I} & -\bar{U} \\ 0 & \mathcal{I} - \bar{V} \end{bmatrix}^{-1} \begin{bmatrix} \bar{A}\bar{H}\bar{F} \\ \bar{B}\bar{H}\bar{F} \end{bmatrix}, \tag{13}$$

which corresponds to a NBGS iteration for the coupled system in Fig. 2. In numerical experiments, we find that the NBGS approach is always faster than a Newton approach, and it is robust except in stiff ODEs, as we later show.

3. Optimizer-based approach In the optimizer-based approach, we treat the ODE state variables as design variables in the larger MDO problem. This is the typical approach in direct-transcription optimal control. This approach requires the formulation of an optimization problem; simply running the model does not result in a converged ODE. In Fig. 3, we see nearly the same dependency graph as we did for the solver-based approach with the exception of the feedback loop. Here, instead of treating Eq. (13) as part of a fixed-point iteration, it is treated as an optimization constraint. Therefore, running the model is much faster than in the solver-based approach, but the optimizer-based approach increases the size of the optimization problem and results in more optimization iterations in most cases.

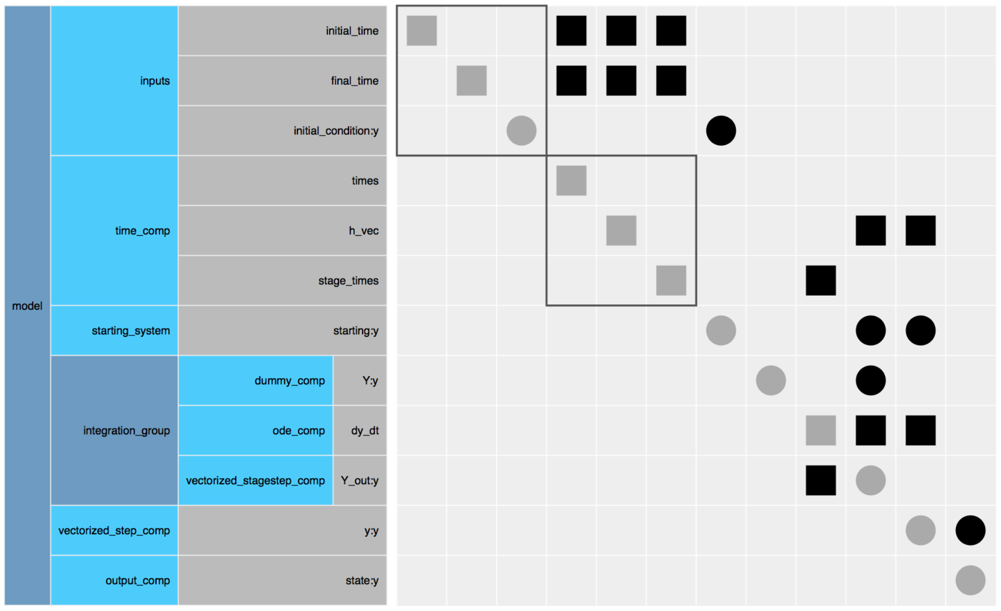
IV. Results

A. Integration methods and verification

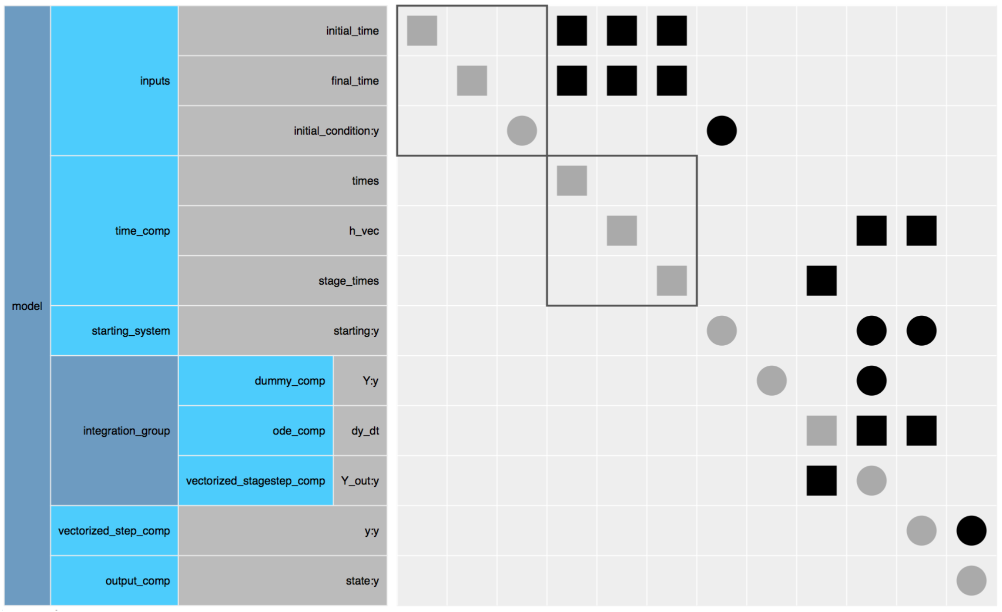
The use of GLMs enables rapid implementation of a large set of integration methods. Figure 4 plots the order of convergence for the implemented integration methods, tested against a simple ODE given by

$$y' = ay \quad , \quad y(0) = 1, \tag{14}$$

where we integrate until $t = 1$ with 10, 15, and 20 time steps.

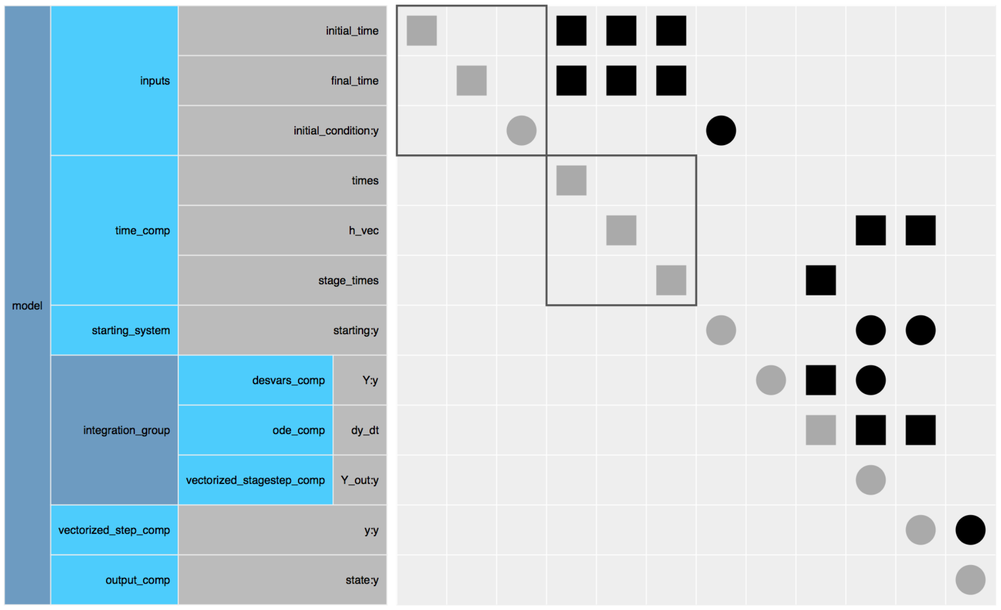


An explicit method: 4th order Runge–Kutta (RK4)

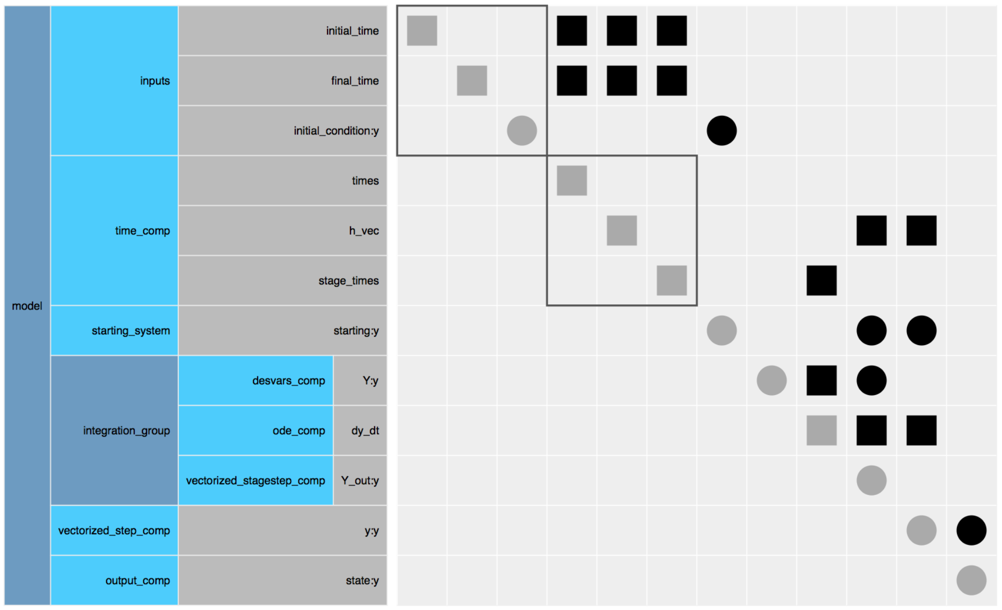


An implicit method: implicit midpoint (IM)

Figure 2: Design structure matrices for the solver-based approach.



An explicit method: 4th order Runge–Kutta (RK4)



An implicit method: implicit midpoint (IM)

Figure 3: Design structure matrices for the optimizer-based approach.

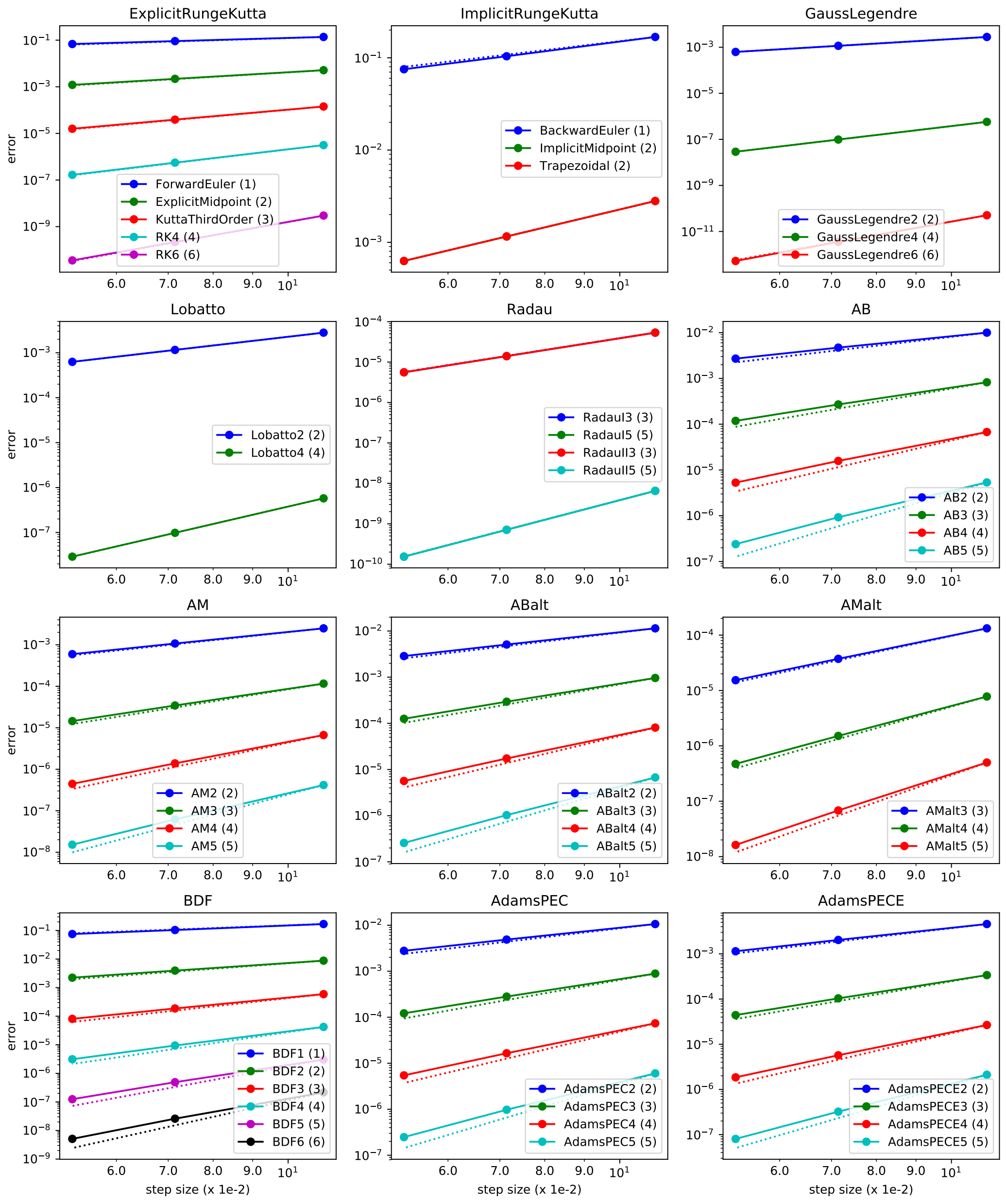


Figure 4: Convergence order plots for various integration methods for a simple homogeneous ODE. In the legend, the expected order is shown in parentheses. AB stands for Adams–Bashforth, AM stands for Adams–Moulton, BDF stands for backward differentiation formula, PEC stands for predict-evaluate-correct, and PECE stands for predict-evaluate-correct-evaluate. The ‘alt’ suffix refers to an alternate implementation of the corresponding method.

The explicit Runge–Kutta methods include forward Euler, explicit midpoint, Kutta's 3rd-order method, the 4th-order Runge–Kutta method, and a 6th-order Runge–Kutta method. The low-order implicit Runge–Kutta methods include backward Euler, implicit midpoint, and the trapezoidal rule. The 2nd, 4th, and 6th order Gauss–Legendre methods are based on the Gauss–Legendre quadrature points. The 2nd and 4th order Lobatto collocation methods contain both endpoints of each time step, and the 3rd and 5th order Radau collocation methods contain the endpoints on only one endpoint of each time step.

The Adams–Bashforth methods are explicit, the Adams–Moulton methods are implicit, and the backward differentiation formulas are also implicit, and all are verified up to at least 5th order. The Adams–Bashforth and Adams–Moulton methods can be implemented in two equivalent ways: one in which there is a single stage and the previous steps' derivatives are included in the step vector, and one in which the previous steps are treated as different stages. The latter is referred to with the 'alt' suffix in Fig. 4. Two types of predictor-corrector methods based on the Adams family are implemented: predict-evaluate-correct and predict-evaluate-correct-evaluate.

For the multi-step methods, a 6th order Runge–Kutta method is used as the starting method to ensure no loss of order. The same formulation (time-marching, solver-based, or optimizer-based) that is used for the multi-step method is used for the starting method.

B. Comparison between formulations

In Fig. 5, we plot ODE solution times versus step size for the time-marching, solver-based, and optimizer-based formulations. The problem is a simple nonlinear ODE given by

$$y' = ty^2 \qquad , \quad y(0) = 1, \tag{15}$$

where we integrate until $t = 1$ with 21, 26, 31, 36 time steps.

The time-marching formulation is the slowest by almost an order of magnitude compared to the solver-based in some cases, and the optimizer-based formulation falls in between the other two. As expected, the computation time for the time-marching formulation consistently increases linearly with the number of time steps. The optimizer-based also shows linear scaling; we expect that this is because the linearly increasing number of design variables causes a linear increase in the number of optimization iterations. However, the solver-based formulation does not show a strong correlation with the number of time steps, and this is due to the vectorization and fast evaluation of the ODE function. Our conclusion is that for simple, non-stiff ODEs that are fast to evaluate, the solver-based formulation results in the fastest ODE solution time.

We expect that if we use the total optimization solution time as the metric in a larger MDO problem, the optimizer-based formulation will perform better compared to the others when it converges. At the same time, for an MDO problem with multiple ODEs with large numbers of time steps, the optimizer-based formulation may be less robust due to the larger design space. Re-running this comparison in an MDO context would be an interesting topic for future work.

C. Comparison between methods

In this section, we compare the integration methods with respect to 3 representative problems: a stiff ODE, a simple non-stiff nonlinear ODE, and an orbital dynamics ODE. They are chosen to represent different types of ODEs and because they all have analytical solutions. We show Pareto fronts for number of function evaluations versus error and total computation time versus error. Our aim is to draw conclusions on the integration methods that provide the best performance in the context of gradient-based MDO where we consider the solver-based and optimizer-based formulations.

The first problem models flame propagation [a], and it is included to represent stiff problems that arise in aerospace applications, e.g., due to ill-conditioned mass matrices. The stiff ODE is given by

$$y' = y^2 - y^3 \qquad , \quad y(0) = \delta, \tag{16}$$

where $\delta = 10^{-2}$ we integrate until $t = 2/\delta$.

The second problem is the simple nonlinear ODE used previously, given by

$$y' = ty^2 \qquad , \quad y(0) = 1, \tag{17}$$

where we integrate until $t = 1$.

[a]From https://www.mathworks.com/company/newsletters/articles/stiff-differential-equations.html

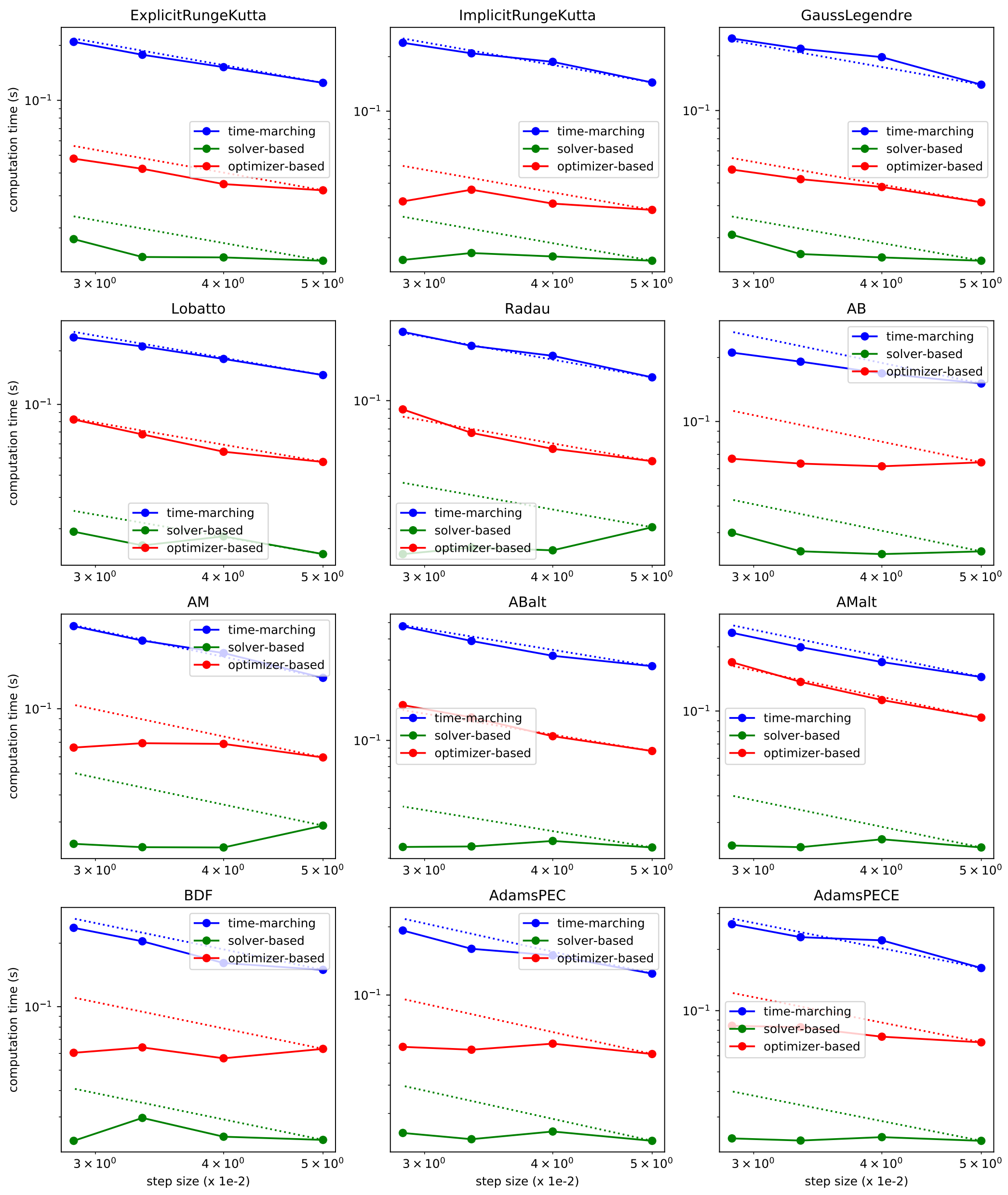


Figure 5: Comparison of the time-marching, solver-based, and optimizer-based formulations with respect to ODE solution time for a simple nonlinear ODE.

The third problem is an orbital dynamics ODE that solves Newton's 2nd law in 2-D. The ODE is given by

$$\begin{bmatrix} r_x' \\ r_y' \\ v_x' \\ v_y' \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ -r_x/r^3 \\ -r_y/r^3 \end{bmatrix} \quad , \quad r_x(0) = 1-e, r_y(0) = 0, v_x(0) = 0, v_y(0) = \sqrt{\frac{1+e}{1-e}}, \tag{18}$$

where $e = 0.5$ and we integrate until $t = 5\pi$, which corresponds to 2.5 revolutions

1. Stiff ODEs

Stiff ODEs represent a significant challenge for many integration methods. Since nearby solutions may vary rapidly, it is often necessary to use very small timesteps to obtain the correct solution. Due to the increased numerical difficulty, Eq. (10) is solved using a Newton-type solver instead of the NBGS solver, which fails to converge in many situations. The optimizer uses the SLSQP optimizer from the SciPy library. We run the ODE solvers with 20, 30, 40, 50, 60, 70, 80, 90, and 100 time steps, and the results are displayed below in Figs. 6, 7, and 8. For brevity, only results with a maximum relative error of 0.1 (or 10%) are displayed.

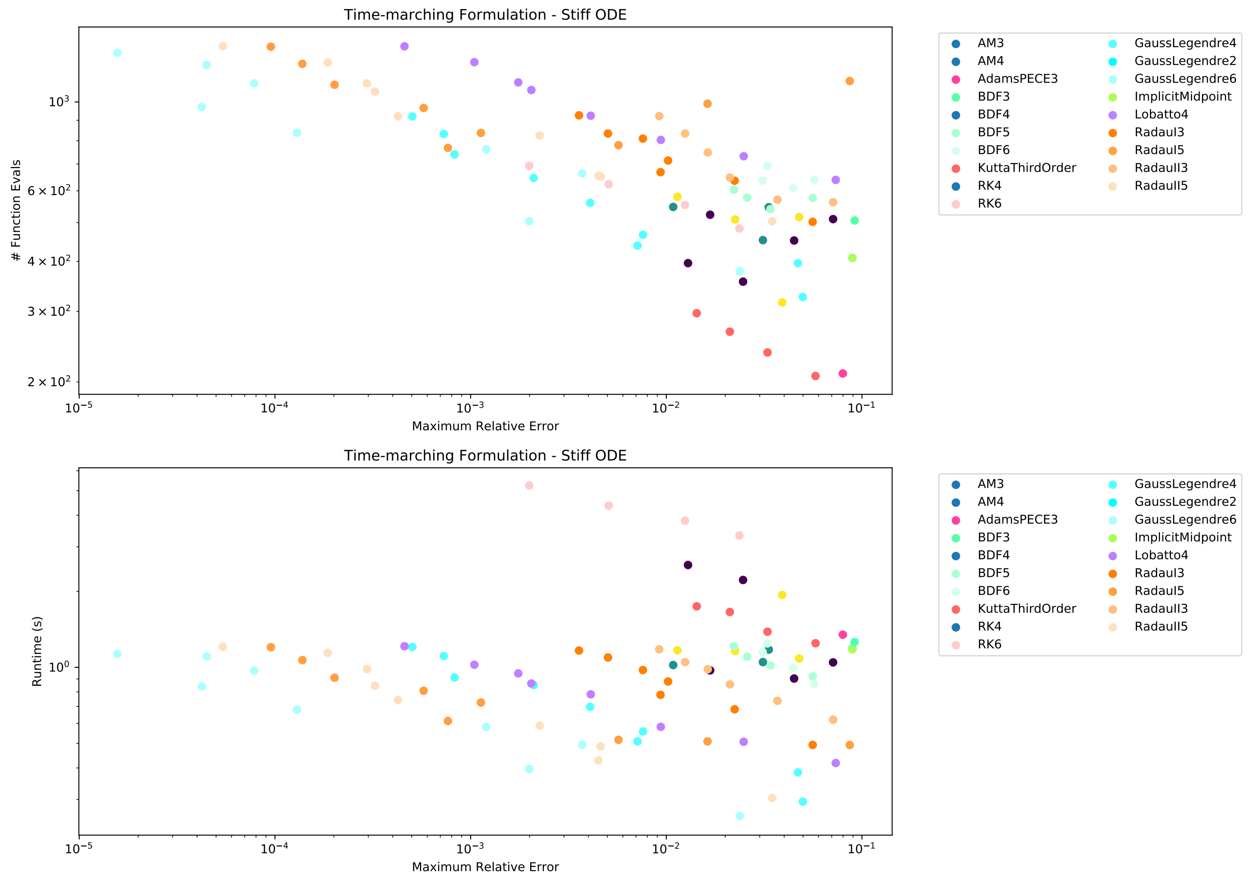
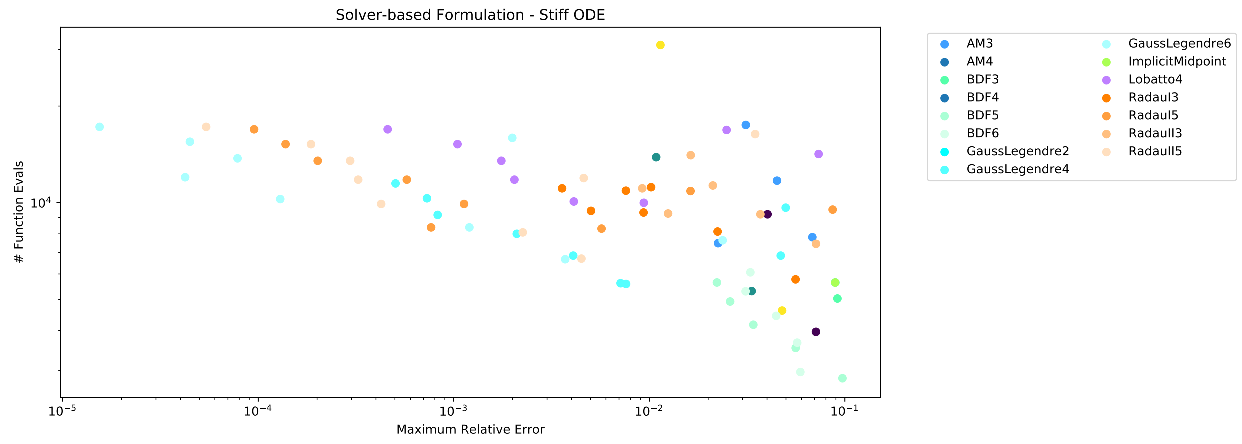


Figure 6: Comparison between methods for the stiff ODE with time-marching.

With this problem, the stiff dynamics proved a significant hardship. Many methods were not able to converge in the vectorized approaches. For example, RK6 was not able to converge in either of the vectorized approaches but ran successfully in the time-marching approach. The Pareto front is composed entirely of implicit methods, as expected. Explicit methods for stiff problems are known to require an excessively small timestep to produce accurate results. In particular, the Gauss–Legendre methods are optimal for higher accuracy requirements. The Radau methods also were high performing, but trailed the Gauss–Legendre methods slightly. For the vectorized approaches, the BDF family



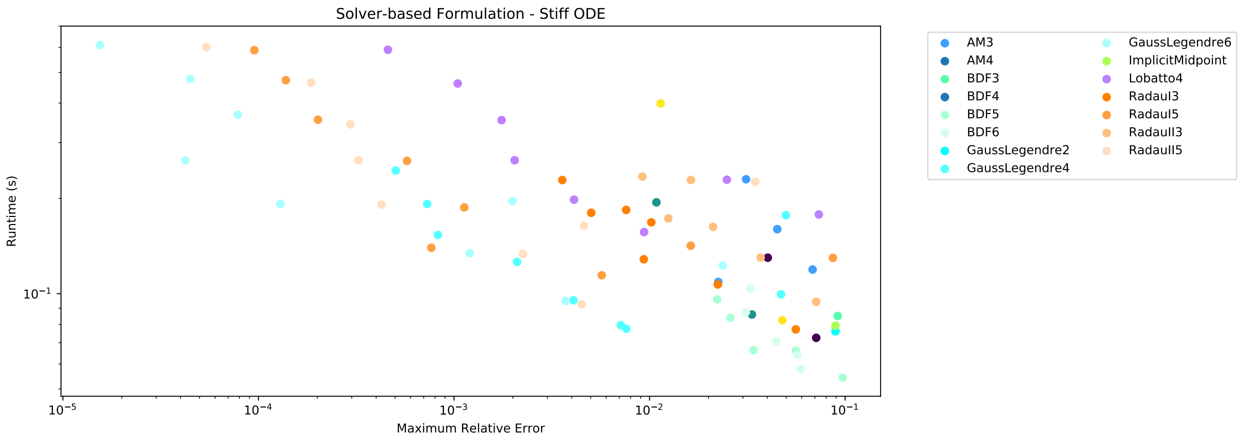


Figure 7: Comparison between methods for the stiff ODE with the solver-based formulation.

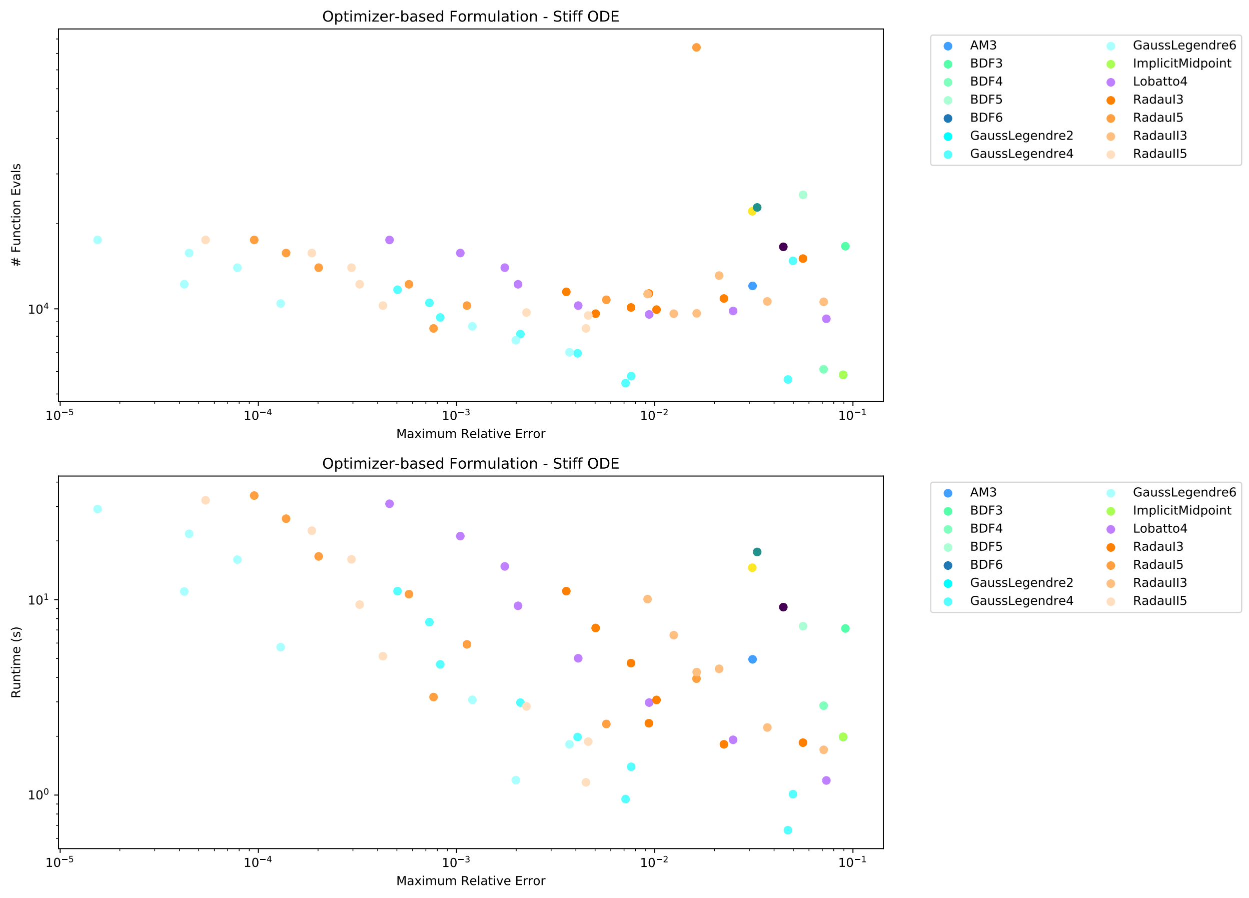


Figure 8: Comparison between methods for the stiff ODE with the optimizer-based formulation.

is suitable for low-mid level accuracy requirements. In the time-marching approach, the predictor corrector method 'AdamsPECE' enjoys an optimal number of function evaluations for low accuracy requirements, but took significantly more time than the Gauss–Legendre methods.

2. *Non-stiff nonlinear ODE*

As in the previous section, the solver-based approach had the highest performance in terms of runtime. For brevity, only the solver-based approach is detailed here. We used 10, 20, 40, and 60 timesteps and again only display methods with less than 0.1 (or 10%) relative error.

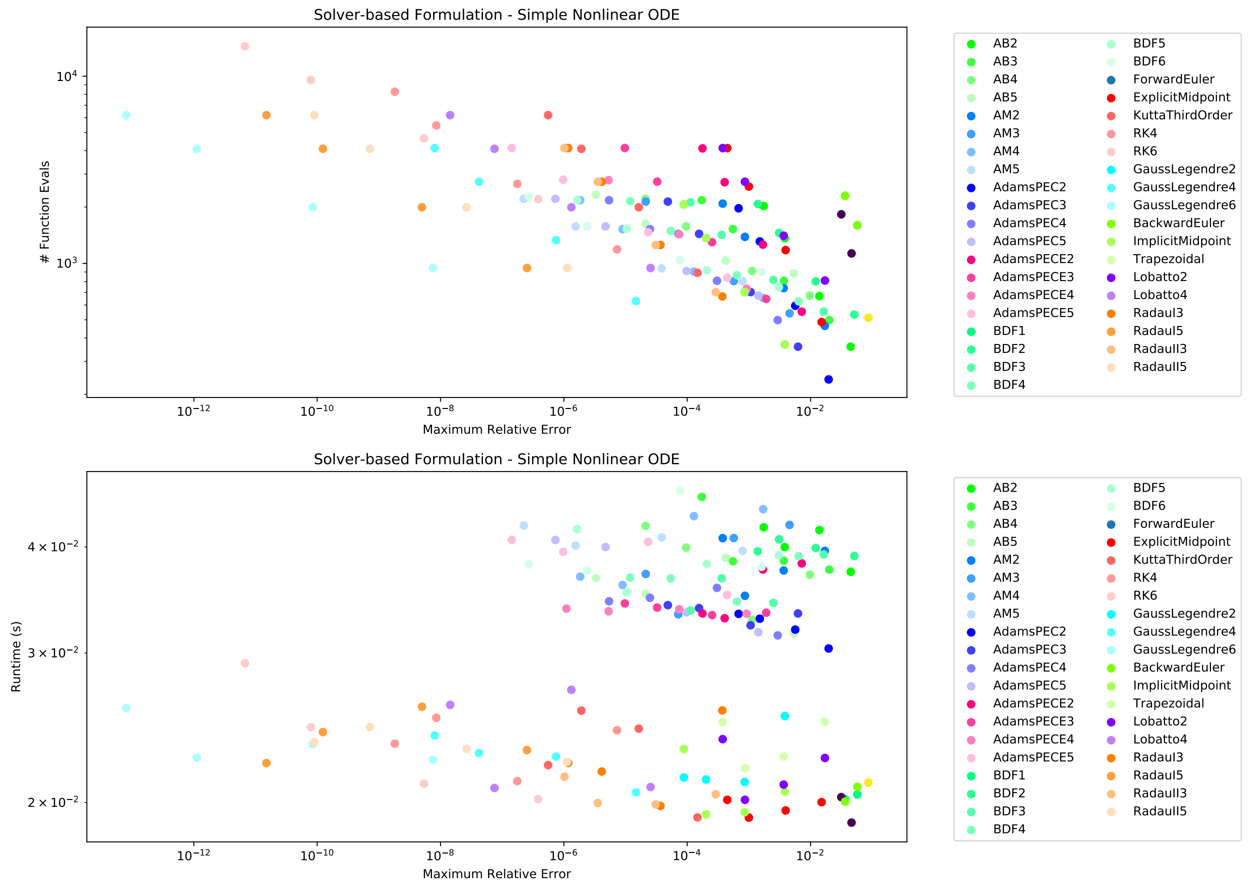


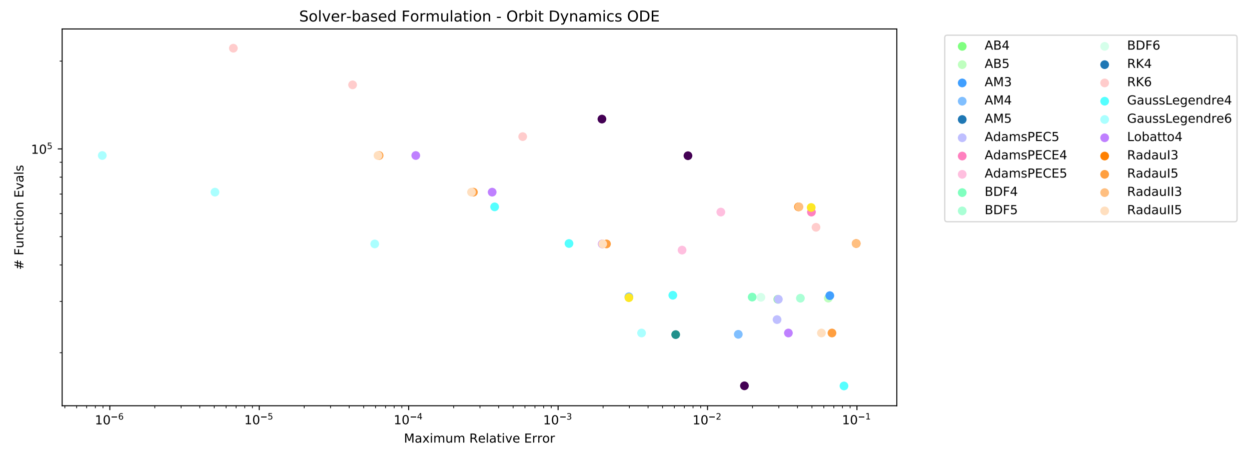
Figure 9: Comparison between methods for the nonlinear ODE with the solver-based formulation.

As before, we see the Gauss–Legendre methods dominate the function evaluation requirement for high level accuracy. Unlike the stiff problem, however, the high-order explicit Runge–Kutta methods become more competitive in the runtime-accuracy front, especially for low- and mid-level accuracy regions. In particular, the explicit midpoint and RK6 methods appear to be optimal for low- and mid-levels, respectively.

3. *Orbital dynamics ODEs*

As in the previous section, the solver-based approach had the highest performance in terms of runtime. For brevity, only the solver-based approach is detailed here. We used 50, 100, 150, and 200 timesteps and again only display methods with less than 0.1 (or 10%) relative error.

The longer time period of this problem highlights an additional feature for the collocation-based methods (e.g. Gauss–Legendre, Radau, Lobatto). While the true solution will preserve certain quantities such as total system energy and angular momentum, the numerical solutions, in general, may not. However, there are methods designed to preserve such invariants and the collocation-based methods will do so [12]. Note that, even with as many as 200 timesteps,



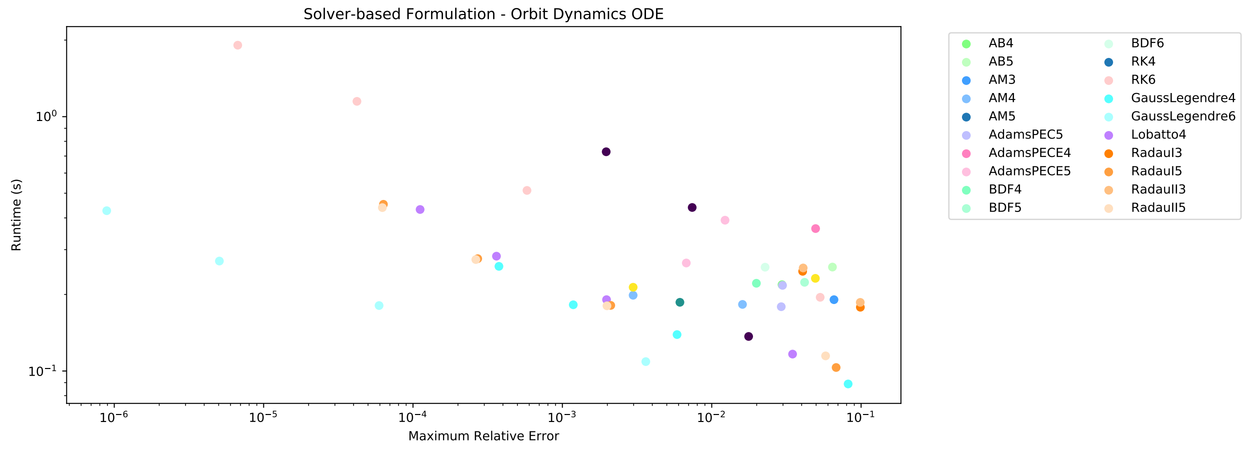


Figure 10: Comparison between methods for the orbital dynamics ODE with the solver-based formulation.

lower order explicit methods were not accurate enough to be shown. This is due to a phenomenon known as energy drift, where the numerical solution's total energy will change over time (explicit methods often increase in energy, implicit methods often decrease in energy). Even higher order implicit methods such as 'BDF6' perform poorly in terms of accuracy. The methods that are capable of preserving these inherent structures perform the best.

In particular, we see the Gauss–Legendre methods dominate the Pareto front for both function evaluations and runtime. The Radau methods and 'RK6' are able to produce higher levels of accuracy, but require many more function evaluations and longer time.

V. Conclusion

In this paper, we presented methods and recommendations for solving ordinary differential equations in gradient-based multidisciplinary design optimization (MDO). Gradient-based MDO presents unique requirements in terms of derivative computation, motivating a modular framework approach.

We identify two contributions in this work. The first is the use of the general linear methods (GLM) formulation, which enables derivatives to be available for any integration method. Any Runge–Kutta or linear multistep method can be added by specifying 4 matrices in the GLM equations, and gradient-based MDO can be performed without having to differentiate the equations for each stage in the method. This enabled the rapid development of close to 50 integration methods and allowed us to run many comparisons across problems.

The second contribution is a new, solver-based, formulation for solving the ODE in the modular gradient-based MDO context. The solver-based formulation performs parallel time integration by treating the ODE equations as a nonlinear system in the GLM form, and using a nonlinear block Gauss–Seidel algorithm. This algorithm iterates between evaluating the vectorized ODE function and the stage-step values for all time steps simultaneously, taking advantage of a factorization that can be performed once as an pre-processing step. The solver-based formulation benefits from vectorization as time-marching is very slow due to framework overhead in a modular gradient-based MDO approach.

We use the GLM approach to perform several benchmarking studies using a stiff ODE, a non-stiff nonlinear ODE, and an orbital dynamics ODE as test problems. Overall, we find that high-order implicit Runge–Kutta methods, especially Gauss–Legendre collocation, are the most robust and efficient across a wide range of problems. Since time-marching is inefficient in this modular gradient-based MDO context, high-order implicit methods perform the most favorably in the orbital dynamics problem, where high-order explicit methods with time-marching are traditionally known as the most performant. Finally, in comparing formulations, we find that the solver-based formulation is the fastest for the majority of problems, with the exception that in stiff ODEs, it fails to converge, and time-marching is the best choice.

As part of this research effort, we have developed an ODE solver library for the OpenMDAO software framework called Ozone. Ozone was used to generate all of the benchmarking results. We expect to make Ozone open-source in the near future.

VI. Acknowledgments

This work was supported by the NASA ARMD Transformational Tools and Technologies Project. The authors would like to thank Robert Falck, Jon Burt, and Justin Gray for insightful discussions.

References

[1] Hwang, J. T., *A modular approach to large-scale design optimization of aerospace systems*, Ph.D. thesis, University of Michigan, 2015.

[2] Martins, J. R. R. A. and Hwang, J. T., "Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models," *AIAA Journal*, Vol. 51, No. 11, November 2013, pp. 2582–2599. doi:10.2514/1.J052184.

[3] Heath, C. and Gray, J., "OpenMDAO: Framework for Flexible Multidisciplinary Design, Analysis and Optimization Methods," *Proceedings of the 53rd AIAA Structures, Structural Dynamics and Materials Conference*, Honolulu, HI, April 2012, AIAA-2012-1673.

[4] Nievergelt, J., "Parallel methods for integrating ordinary differential equations," *Communications of the ACM*, Vol. 7, No. 12, 1964, pp. 731–733.

[5] Gander, M. J., "50 years of time parallel time integration," *Multiple Shooting and Time Domain Decomposition Methods*, Springer, 2015, pp. 69–113.

[6] Bellen, A. and Zennaro, M., “Parallel algorithms for initial-value problems for difference and differential equations,” *Journal of Computational and applied mathematics*, Vol. 25, No. 3, 1989, pp. 341–350.

[7] Chartier, P. and Philippe, B., “A parallel shooting technique for solving dissipative ODE’s,” *Computing*, Vol. 51, No. 3, 1993, pp. 209–236.

[8] Lions, J.-L., Maday, Y., and Turinici, G., “A parareal in time discretization of PDEs,” *Comptes Rendus de l’Academie des Sciences. Series I, Mathematics*, Vol. 332, No. 7, 2001, pp. 661–668.

[9] Topputo, F. and Zhang, C., “Survey of direct transcription for low-thrust space trajectory optimization with applications,” *Abstract and Applied Analysis*, Vol. 2014, Hindawi Publishing Corporation, 2014.

[10] Butcher, J. C., “General linear methods,” *Acta Numerica*, Vol. 15, 2006, pp. 157–256.

[11] Burrage, K. and Butcher, J. C., “Non-linear stability of a general class of differential equation methods,” *BIT Numerical Mathematics*, Vol. 20, No. 2, 1980, pp. 185–203.

[12] Hairer, E., Lubich, C., and Wanner, G., *Geometric Numerical Integration*, Vol. 31 of *Springer Series in Computational Mathematics*, Springer-Verlag, Berlin/Heidelberg, 2nd ed., 2006. doi:10.1007/3-540-30666-8.