SIAM J. SCI. COMPUT.
Vol. 41, No. 5, pp. A2911–A2937

CONSTRUCTING NEW TIME INTEGRATORS USING INTERPOLATING POLYNOMIALS*

TOMMASO BUVOLI† AND MAYYA TOKMAN‡

Abstract. We present a methodology for constructing time integrators for solving systems of first-order ordinary differential equations by using interpolating polynomials. Our approach is to combine ideas from complex analysis and approximation theory to construct new integrators. This strategy allows us to trivially satisfy order conditions and easily construct a range of implicit or explicit integrators with properties such as parallelism and high order of accuracy. In this work, we present several example polynomial methods including generalizations of the backward differentiation formula and Adams–Moulton methods. We compare the stability regions of these generalized methods to their classical counterparts and find that the new methods offer improved stability especially at high order.

Key words. time integration, polynomial interpolation, approximation theory, parallelism, high-order

AMS subject classifications. 65L04, 65L05, 65L06, 65E99

DOI. 10.1137/18M1203808

1. Introduction.

Due to their simplicity and adaptability, polynomials are widely used in approximation theory to estimate functions, derivatives and integrals. The interpolating polynomial, which matches function and derivative values at a set of nodes, is one of the most commonly used approximations. By choosing stable interpolation nodes it is even possible to construct convergent interpolating polynomials of very high degree [21, 5].

The use of interpolating polynomials has led to several important developments in the solution of partial differential equations, most notably finite difference methods [18] and spectral methods [10, 20]. In each case, a polynomial approximation, either local or global, is formed using spatial information and then differentiated. Interpolating polynomials have also been applied in the time dimension to derive linear multistep methods such as backward differentiation formulas (BDFs), Adams–Moulton (AM) methods, and Adams–Bashforth methods.

When discussing polynomial approximations, it is natural to consider the complex plane, especially for numerical differentiation. Sampling a function off the real line helps overcome the ill-conditioning inherent in classical finite difference formulas for high derivatives. For example, by using polynomial interpolation at the roots of unity, Lyness and Fornberg both developed perfectly well-conditioned algorithms for numerical differentiation [9, 15].

The complex plane has also proven invaluable in other mathematical domains. In the context of time discretization, there have been several attempts to study classical methods along complex-valued integration paths. Examples include Taylor methods

*Submitted to the journal's Methods and Algorithms for Scientific Computing section October 16, 2018; accepted for publication (in revised form) June 26, 2019; published electronically October 1, 2019.

https://doi.org/10.1137/18M1203808

Funding: This work was funded by the National Science Foundation, Computational Mathematics Program, under DMS-1115978 and DMS-1216732.

†Department of Applied Mathematics, University of Washington, Seattle, WA 98195 (buvoli@uw.edu).

‡School of Natural Sciences, University of California, Merced, CA 95343 (mtokman@ucmerced.edu).

in the complex plane [8], a Taylor–Padé ODE solver for the Painlevé equations [11], and an analysis of superconvergence for classical Runge–Kutta methods along complex paths of integration [16]. Another interesting attempt to incorporate complex time was presented in [14], where complex coefficients enabled the construction of high-order splitting schemes for analytic semigroups.

In this work, we utilize polynomial approximations in the complex plane to construct new classes of time integrators that retain the derivational simplicity of linear multistep methods while improving upon their stability restrictions. Our broad aim is to introduce a new construction strategy for deriving stable, high-order integrators that provides a viable alternative to existing approaches such as extrapolation methods, spectral deferred correction methods, and fully implicit collocation methods.

Our methodology for constructing time integrators using polynomials has three principal components:

1. *Complex time integration.* Our methodology allows for the development of integrators that compute inputs and stages at complex times. This generalization allows us to construct schemes that more broadly incorporate ideas from approximation theory and stable numerical differentiation. The novelty of our approach is in the direct use of complex temporal nodes to design classical-type time integrators with improved stability properties.
2. *A new parameter.* To construct polynomial integrators we introduce an additional parameter that we call the extrapolation factor. This new parameter is essential for obtaining methods with improved stability regions.
3. *A new method formulation.* We describe methods in terms of the interpolating polynomials used to compute stage and output values. This formulation is complementary to the traditional coefficient representation.

The paper is organized as follows. Section 2 introduces ordinary differential equations (ODEs) in the complex time plane and discusses an explicit polynomial method that can be derived from either a discretized Cauchy integral formula or polynomial interpolation. In section 3 we introduce our new polynomial-based construction approach and derive polynomial block methods (PBMs). We discuss the order of accuracy for these integrators and provide a simple procedure for computing coefficients. In section 4 we present several strategies for constructing PBMs and introduce two schemes that generalize backward differentiation and AM methods. Finally in section 5 we present several numerical experiments for these generalized methods and compare them to their classical counterparts.

1.1. The model problem. We are interested in developing time integrators for solving systems of ODEs of the form

$$\begin{aligned} &\frac{dy_j}{dt} = F_j(t, y_1, \ldots, y_M), && j = 1, \ldots, M. \\ &y_j(t_n) = |y_{j,n}| < \infty, \end{aligned} \tag{1.1}$$

To improve the clarity of our presentation we introduce all of our methods using the scaler equation

$$y'(t) = F(t, y(t)), \qquad y(t_n) = y_n. \tag{1.2}$$

However, all of our results are directly applicable to the nonscalar case.

2. Time-stepping in the complex plane. In this section we briefly discuss how initial value problems posed on a real time interval can be extended into the complex time plane. We then use this observation to construct an explicit time-stepping method that computes the solution at complex time points. We first derive this method using the Cauchy integral formula and then present an alternative derivation using Lagrange interpolating polynomials in the complex plane. Interpreting this method from a polynomial perspective provides a clear direction for further generalization and motivates the construction of a wide class of time integrators based on interpolating polynomials.

2.1. Ordinary differential equations in complex time. It is normally assumed that the initial value problem (1.2) is valid only for real time. However, if the solution $y(t)$ is locally analytic in a region containing the initial condition, then we may extend the initial value problem to complex time via analytic continuation. Local analyticity is guaranteed by the following well-known theorem.

THEOREM 2.1 (Cauchy–Kowalevski). *Consider the system of ODEs* (1.1). *If each* $F_j(t, y_1, \ldots, y_M)$, $j = 1, \ldots, M$, *is an analytic function of each of its arguments in a domain* D *containing* $t = t_n$, *then* (1.1) *has a unique, analytic solution in a neighborhood of* t_n [1, section 3.7].

The conditions required by the Cauchy–Kowalevski theorem include many common ODEs and spatial discretizations of partial differential equations. If we restrict ourselves to these problems, it is possible to explore time integration methods in the complex time plane.

Using a Taylor expansion we can express the solution in complex time $t = t_n + h$ via

$$y(t_n + h) = \sum_{\nu=0}^{q} \frac{y^{(\nu)}(t_n) h^\nu}{\nu!} + \mathcal{O}(h^{q+1}). \tag{2.1}$$

Despite the simplicity of a Taylor series method, it is generally impractical to differentiate the right-hand side of (1.1) explicitly due to complexity. As an alternative to manual differentiation, we propose to utilize the Cauchy integral formula coupled with trapezoidal quadrature to approximate derivatives of $y(t)$. This approximation is well-conditioned, is spectrally accurate, and allows for simple calculation of high-order derivatives [15, 9, 3, 4, 2]. This strategy will lead us naturally to a multivalue time integrator that approximates the solution at complex time points.

2.2. A time-stepping scheme based on the Cauchy integral formula. We now derive a new explicit time integrator based on the Cauchy integral formula. This new method applies techniques from stable numerical differentiation [15, 9] to the time domain and joins a host of other algorithms that use analytic functions at the roots of unity [2].

By restricting ourselves to the class of initial value problems that satisfy the conditions of the Cauchy–Kowalevski theorem, we are guaranteed that there exists some $R > 0$ such that the ODE solution $y(t)$ is analytic inside the circular domain

of radius R centered around t_n. This allows us to express the local derivatives of the solution at $t = t_n$ using the Cauchy integral formula:

$$y^{(0)}(t_n) = \frac{1}{2\pi i} \oint_\Gamma \frac{y(z)}{(z-t_n)} dz,$$
$$y^{(\nu \geq 1)}(t_n) = \frac{(\nu-1)!}{2\pi i} \oint_\Gamma \frac{y'(z)}{(z-t_n)^\nu} dz = \underset{\textbf{Apply ODE}}{\uparrow} \frac{(\nu-1)!}{2\pi i} \oint_\Gamma \frac{F(z,y(z))}{(z-t_n)^\nu} dz,$$

where Γ is a simple closed contour inside the disk of radius R enclosing t_n. If we take Γ to be a circular contour of radius $r < R$ centered at t_n and apply the change of variables $z = re^{i\theta} + t_n$, then the contour integrals reduce to

$$y^{(0)}(t_n) = \frac{1}{2\pi} \int_0^{2\pi} y(re^{i\theta} + t_n) d\theta,$$
$$y^{(\nu \geq 1)}(t_n) = \frac{(\nu-1)!}{2\pi r^{\nu-1}} \int_0^{2\pi} \frac{F(re^{i\theta} + t_n, y(re^{i\theta} + t_n))}{e^{i(\nu-1)\theta}} d\theta.$$

The change of variables leads to locally analytic, periodic integrands. By discretizing the interval $[0, 2\pi)$ into the w points $\theta_j = \frac{2\pi(j-1)}{w}$, $j = 1, \ldots, w$, we may approximate the integrals to exponential accuracy using the trapezoidal rule [22]. This leads us to the approximations

$$y^{(0)}(t_n) \approx \sum_{j=1}^{w} \frac{y_j^{[n]}}{w}, \qquad y^{(\nu \geq 1)}(t_n) \approx \frac{(\nu-1)!}{w r^{\nu-1}} \sum_{j=1}^{w} f_j^{[n]} e^{-i(\nu-1)\theta_j}, \tag{2.2}$$

where $y_j^{[n]}$ and $f_j^{[n]}$ are the solution and derivative values at the scaled roots of unity enclosing $t = t_n$ such that

$$\left.\begin{aligned} y_j^{[n]} &= y(t_j^{[n]}) \\ f_j^{[n]} &= F(t_j^{[n]}, y_j^{[n]}) \\ t_j^{[n]} &= re^{i\theta_j} + t_n \end{aligned}\right\} \quad j = 1, \ldots, w.$$

We may now substitute the derivative approximations (2.2) into the Taylor polynomial (2.1). In order to develop a time-stepping method, we must evaluate the polynomial at the points $t_j^{[n+1]} = t_j^{[n]} + h$. The timestep iteration can be written as

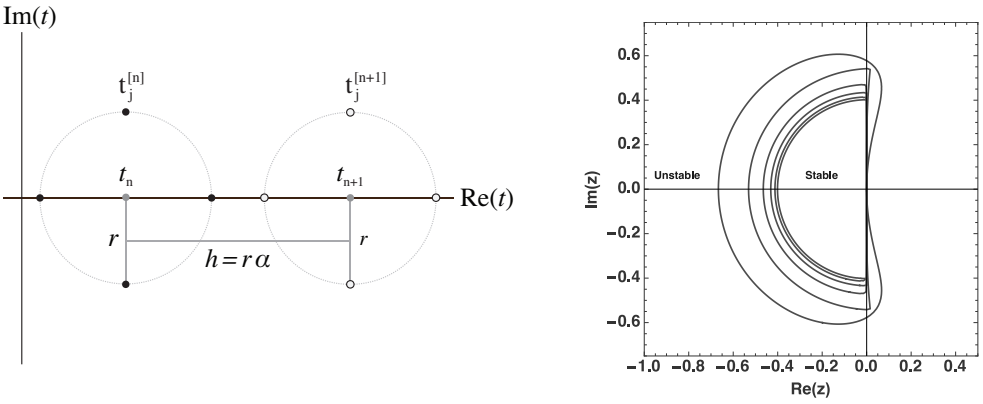
$$y_j^{[n+1]} = \frac{1}{w} \sum_{k=1}^{w} y_k^{[n]} + \sum_{\nu=1}^{q} \left[\frac{\left(h + re^{i\theta_j}\right)^\nu}{\nu w r^{\nu-1}} \sum_{k=1}^{w} f_k^{[n]} e^{i(1-\nu)\theta_k} \right], \qquad j = 1, \ldots, w.$$

It is convenient to introduce the *extrapolation factor* α and parametrize the stepsize h as the product of α and the radius r. By letting $h = r\alpha$ and taking $w = q$, the timestep simplifies to

$$y_j^{[n+1]} = \frac{1}{q} \sum_{k=1}^{q} y_k^{[n]} + r \sum_{\nu=1}^{q} \sum_{k=1}^{q} \left[\frac{\left(\alpha + e^{i\theta_j}\right)^\nu e^{i(1-\nu)\theta_k}}{\nu q} \right] f_k^{[n]}, \qquad j = 1, \ldots, q. \tag{2.3}$$

In Figure 1(a) we present a graphical representation of the input and output times $t_j^{[n]}$, $t_j^{[n+1]}$ and the parameters r, h and α for $q = 4$. At the first timestep the method (2.3) requires the solution values at each of the q roots of unity.

The time-stepping scheme (2.3) is a Taylor method where the derivatives have been approximated by discrete Cauchy integrals. This explicit time integrator has several interesting properties. First the derivation is simple to understand and the method is easy to implement for any order of accuracy by varying q. Second, unlike many explicit linear multistep methods, the linear stability regions do not contract to zero at least up to order seven (See Figure 1(b)). Numerical results suggest that this property holds for higher-order methods as well, although we have not proved this theoretically. Finally, the method is parallel since each of the q right-hand-side evaluations may be computed simultaneously at each timestep.



(a) (b)

FIG. 1. (a) *Input and output times for the method* (2.3) *with* $q = 4$. *Input times* $t_j^{[n]}$ *are labeled with black circles, output times* $t_j^{[n+1]}$ *are labeled with white circles, and the timestep centers* t_n, $t_{n+1} = t_n + h$ *are labeled with gray circles. Two circles of radius* r *centered around* t_n *and* t_{n+1} *are also plotted. The stepsize* h *has been parametrized as* $h = r\alpha$, *where* α *denotes the number of radii* r *per timestep.* (b) *Numerically computed linear stability regions of the method* (2.3) *for* $\alpha = 1$ *and* $q = 2, 3, \ldots, 7$. *The stability regions contract monotonically as* q *increases. Methods with* $q = 2, 3, 6,$ *and* 7 *are stable along an imaginary interval containing the origin.*

2.3. An alternative derivation using polynomial interpolation. We derived the integrator (2.3) by considering a truncated Taylor series where the exact derivatives were replaced with approximations computed using a discrete Cauchy integral formula. It is natural to ask if we may generalize this idea by considering other approximations to derivatives. We are motivated by the observation that the discrete Cauchy integral formula we used to derive (2.3) produces the same approximations as the polynomial-based finite difference formula with nodes at the roots of unity. We will briefly explore this connection and then rederive the integrator (2.3) using polynomials.

Remark 2.2. Suppose we seek to approximate the derivatives of a function $g(z)$ at $z = 0$ using the values $g_k = g(z_k)$, where z_k are scaled roots of unity given by

$$z_k = r \exp(2\pi i(k-1)/q) \qquad \text{for} \qquad k = 1, \ldots, q.$$

Then, the following two algorithms are equivalent.

Algorithm 1. *Polynomial differentiation.* Differentiate a Lagrange interpolating polynomial $p(z)$ that passes through each g_k, then $g^{(\nu)}(0) \approx \hat{g}^\nu = p^{(\nu)}(0)$.

Algorithm 2. *Discrete Cauchy integral formula.* Approximate the Cauchy integral formula using a circular contour or radius r centered at zero and trapezoidal quadrature at each of the points z_k, then

$$g^{(\nu)}(0) = \frac{\nu!}{2\pi i} \oint_\Gamma \frac{g(\zeta)}{\zeta^{\nu+1}} \approx \tilde{g}^\nu = \frac{\nu!}{qr^\nu} \sum_{k=1}^{q} g_k e^{-i\nu\theta_k} \qquad \text{for} \qquad \theta_k = \frac{2\pi(k-1)}{q}. \tag{2.4}$$

To prove the equivalence of the two algorithms, i.e., $\hat{g}^\nu = \tilde{g}^\nu$, we express the Lagrange interpolating polynomial as $p(z) = \sum_{k=0}^{q-1} a_k z^k$, where the coefficients a_k are obtained by solving the $q \times q$ linear system

$$\mathbf{WD}\,\mathbf{a} = \mathbf{g} \qquad \text{for} \qquad \begin{aligned} \mathbf{W}_{ij} &= (z_j/r)^{i-1}, \\ \mathbf{D} &= \operatorname{diag}(1, r, \dots r^{q-1}), \end{aligned} \qquad \text{and} \qquad \begin{aligned} \mathbf{a} &= [a_0, \dots, a_{q-1}]^T, \\ \mathbf{g} &= [g_1, \dots, g_q]^T. \end{aligned}$$

The Vandermonde matrix $\mathbf{W}$ is the inverse discrete Fourier transform matrix. We can obtain an explicit formula for $\mathbf{a}$ by inverting the system, yielding

$$\mathbf{a} = \mathbf{D}^{-1}\mathbf{W}^{-1}\mathbf{g} \qquad \text{for} \qquad \mathbf{W}^{-1}_{ij} = q^{-1}(z_j/r)^{1-i}.$$

Finally, using $p^{(\nu)}(0) = \nu! a_\nu$, we obtain the approximation (2.4). Had we approximated the derivatives at $z \neq 0$, then the trapezoidal approximation of the Cauchy integral formula amounts to rational approximation of the underlying function and the two algorithms would no longer be identical [2].

Using this result we can present a second derivation for the method (2.3) based on interpolating polynomials in the complex plane. At each timestep, we can use the inputs and input derivatives to form local Lagrange interpolating polynomials approximating $y(t)$ and $F(t, y(t))$. We may express these polynomials as

$$y^{[n]}_{\mathcal{P}}(t) = \sum_{j=1}^{q} \ell_j(t) y_j^{[n]} \quad \text{and} \quad F^{[n]}_{\mathcal{P}}(t) = \sum_{j=1}^{q} \ell_j(t) f_j^{[n]}, \quad \text{where} \quad \ell_j(z) = \prod_{\substack{l=1 \\ l \neq j}}^{q} \frac{z - z_l}{z_j - z_l}.$$

Next, we can approximate the derivatives of $y(t)$ at $t = t_n$ as

$$y^{(0)}(t_n) = y^{[n]}_{\mathcal{P}}(0) \qquad \text{and} \qquad y^{(\nu \geq 1)}(t_n) = \frac{d^{\nu-1}}{dt^{\nu-1}} F^{[n]}_{\mathcal{P}}(t)\Big|_{t=0}.$$

By substituting these derivative approximations into the Taylor series (2.1), and evaluating at each $t_j^{[n+1]}$ we obtain the method (2.3).

Interpreting the integrator (2.3) as a polynomial-based method opens many avenues for further generalization. First, we can construct methods that use different node sets and we can consider polynomial approximations which pass through both function and derivative values. Second, we can expand the Taylor series (2.1) at a different point and approximate each derivative using a different polynomial approximation. Finally, we can also consider implicit methods and methods that compute stage values. The framework introduced in the next section incorporates each of these generalizations and allows us to easily construct a wide range of polynomial integrators.

3. Polynomial time integrators. In this section we generalize the ideas presented in section 2 and introduce a methodology for deriving polynomial based time integrators for solving systems of first-order ODEs. Our methodology relies on the use of interpolating polynomials to trivially satisfy order conditions. This simple and flexible approach can be used to construct a wide variety of integrators with favorable properties including those with parallelism and high order of accuracy.

We begin this section by introducing a family of polynomials designed to approximate the solution of initial value problems. We call these polynomials *ODE polynomials*, and we construct them using a corresponding *ODE dataset* that contains approximate solution and derivative data. ODE polynomials form the basis of our methodology but can be regarded more generally as tools for locally approximating the solution to initial value problems. Once we present ODE polynomials and ODE datasets, we proceed by formally introducing polynomial-based time integrators along with their corresponding notation and parameters. We then use this notation to present PBMs and close the section by discussing their linear stability and order of accuracy.

3.1. The ODE dataset. We define an ODE dataset to be an ordered set that contains approximate solution values and derivative values along with their corresponding temporal nodes. The nodes are expressed in the local coordinates τ, where the global time t is given by

$$t(\tau) = r\tau + s. \tag{3.1}$$

All ODE datasets are parametrized by the scaling factor r and the translation constant s; together, these parameters determine the temporal locations of each data element. The scaling factor r is a strictly positive real number, while both τ and s can be complex.

DEFINITION 3.1 (ODE dataset). *An ODE dataset of size w is an ordered set of tuples of the form $D(r,s)=\{(\tau_j,\ y_j,\ rf_j)\}_{j=1}^{w}$ where $y_j \approx y(t(\tau_j))$, and $f_j = F(t(\tau_j), y_j)$.*

Note that each derivative term in the dataset is multiplied by a factor of r to reflect the transformation into local coordinates τ where

$$\frac{d}{dt} = \frac{1}{r}\frac{d}{d\tau}.$$

It is sometimes convenient to categorize the elements of a dataset. We will do so using the following notation:

$$D(r,s) = \begin{cases} \text{data label 1} : \{(\tau_j,\ y_j,\ rf_j)\}_{j\in\mathcal{A}_1} \\ \text{data label 2} : \{(\tau_j,\ y_j,\ rf_j)\}_{j\in\mathcal{A}_2} \\ \qquad\vdots \\ \text{data label n} : \{(\tau_j,\ y_j,\ rf_j)\}_{j\in\mathcal{A}_n}, \end{cases}$$

where the sets $\mathcal{A}_j$ each contain some subset of the indices $1,\ldots,w$, and the union of the sets must produce all the indices from 1 to w. In short, each element in the dataset must pertain to at least one category. In this paper we will frequently categorize the elements of ODE datasets into inputs, outputs, and stage values.

3.2. The ODE polynomial. We define an ODE polynomial to be a polynomial that approximates the solution of an initial value problem and is constructed using values from an ODE dataset. All ODE polynomials are truncated Taylor series of $y(t)$ in which every exact derivative has been replaced with an approximation obtained by differentiating an interpolating polynomial.

If the ODE dataset $D(r, s)$ is used to construct an ODE polynomial, then the polynomial is also expressed in terms of the local coordinates τ where $t(\tau) = r\tau + s$. All ODE polynomials are written in terms of an expansion point b and approximate the truncated Taylor series for the local solution $y(t(\tau))$, expanded around the point $\tau = b$. Regardless of the value of b, every ODE polynomial approximates the local solution $y(t(\tau))$ so that

$$y(t(\tau)) \approx p(\tau; b) \quad \forall b.$$

We formally define an ODE polynomial as follows.

DEFINITION 3.2 (ODE polynomial). *An ODE polynomial of degree g is a polynomial of the form*

$$p(\tau; b) = \sum_{j=0}^{g} \frac{a_j(b)(\tau - b)^j}{j!}, \tag{3.2}$$

where each approximate derivative $a_j(b)$ must be computed using values from an ODE dataset $D(r, s) = \{(\tau_j,\ y_j,\ rf_j)\}_{j=1}^{w}$ in one of the following ways:

1. *By differentiating a polynomial $h_j(\tau)$ that approximates $y(t(\tau))$. $h_j(\tau)$ must be the polynomial of lowest degree that interpolates at least one solution value in the ODE dataset $D(r, s)$ and whose derivative $h_j'(\tau)$ interpolates any number of derivative values in $D(r, s)$ so that*

$$\begin{aligned} h_j(\tau_k) &= y_k \qquad \text{for} \qquad k \in \mathcal{A}^j \text{ and } \mathcal{A}^j \neq \emptyset, \\ h_j'(\tau_k) &= rf_k \qquad \text{for} \qquad k \in \mathcal{B}^j, \end{aligned}$$

   and the sets $\mathcal{A}^j$ and $\mathcal{B}^j$ contain unique indices ranging from 1 to w. Then, the approximate derivative $a_j(b)$ is

$$a_j(b) = \left. \frac{d^j}{d\tau^j} h_j(\tau) \right|_{\tau=b}.$$

2. *By differentiating a polynomial $l_j(\tau)$ that approximates $ry'(t(\tau))$. $l_j(\tau)$ must be the polynomial of lowest degree that interpolates at least one derivative value in the ODE dataset $D(r, s)$ such that*

$$l_j(\tau_k) = rf_k \qquad \text{for} \qquad k \in \mathcal{C}^j \text{ and } \mathcal{C}^j \neq \emptyset,$$

   and the set $\mathcal{C}^j$ contains unique indices ranging from 1 to w. Then, the approximate derivative $a_j(b)$ is

$$a_j(b) = \left. \frac{d^{j-1}}{d\tau^{j-1}} l_j(\tau) \right|_{\tau=b} \qquad \text{(only valid for } j \geq 1\text{)}.$$

It follows from the definition that all ODE polynomials depend implicitly on the scaling factor r. However, we will avoid writing $p(\tau, r; b)$ since the dependance on r does not arise from the structure of the polynomial, but instead comes from the values in the ODE dataset $D(r, s)$ used to compute the derivative terms $a_j(b)$.

Remark 3.3. If an ODE dataset contains nondistinct nodes τ_i (i.e., more than one approximation of $y(t)$ is provided at the same temporal node), then any interpolating polynomial $h_j(\tau)$ or $l_j(\tau)$ that matches solution or derivative information at these nodes must pass through a linear combination of the corresponding solution or derivative values where each of the weights sums to one.

3.2.1. Special families of ODE polynomials. The family of all ODE polynomials is large and the Taylor formulation (3.2) provides no guidance for choosing derivative approximations. To help remedy this problem, we introduce two special families of ODE polynomials that are inspired by classical linear multistep methods.

Let $p(\tau; b)$ be an ODE polynomial of degree g of the form (3.2), constructed from an ODE dataset $D(r, s)$. Then, we make the following statements:

1. The polynomial $p(\tau; b)$ is an *Adams ODE polynomial* if each of its approximate derivatives $a_j(b)$ is formed via

$$a_j(b) = \begin{cases} L_y(b), & j = 0, \\ \frac{d^{j-1}}{d\tau^{j-1}} L_F(\tau)\Big|_{\tau=b,} & j > 0, \end{cases} \tag{3.3}$$

   where (1) $L_y(\tau)$ is a Lagrange interpolating polynomial that interpolates any subset of the solution values in $D(r, s)$, and (2) $L_F(\tau)$ is a Lagrange interpolating polynomial that interpolates at least one derivative value in $D(r, s)$. Any Adams ODE polynomial may be expressed in integral form as

$$p(\tau; b) = L_y(b) + \int_b^\tau L_F(\xi) d\xi, \tag{3.4}$$

   where the expansion point b serves as a left integration endpoint. We name these approximations Adams ODE polynomials since the Adams–Bashforth and AM methods can be derived using an Adams ODE polynomial constructed from an ODE dataset with equispaced nodes.
2. The polynomial $p(\tau; b)$ is a *BDF ODE polynomial* if each of its approximate derivatives $a_j(b)$ is formed via

$$a_j(b) = \frac{d^j}{d\tau^j} H_y(\tau)\Big|_{\tau=b}, \tag{3.5}$$

   where $H_y(\tau)$ is an interpolating polynomial of degree g that interpolates g solution values in $D(r, s)$ and whose derivative $\frac{d}{dt} H_y(\tau)$ interpolates a single derivative value in $D(r, s)$. BDF ODE polynomials do not depend on the expansion point b since

$$p(\tau; b) = H_y(\tau) \quad \forall b.$$

   We name these approximations BDF ODE polynomials since the classical BDF method may be derived from a BDF ODE polynomial constructed from an ODE dataset with equispaced nodes.

3.2.2. Important properties of ODE polynomials. The following properties of ODE polynomials are useful for characterization and method construction:

1. *Degree.* The degree of $p(\tau; b)$ in τ.
2. *Truncation error.* The truncation error for $p(\tau; b)$ is given by

$$\mathrm{TE}(r, \tau) = \left| \tilde{p}\,(\tau; b) - y(t(\tau)) \right|,$$

   where $\tilde{p}(\tau; b)$ is the ODE polynomial we would obtain had we replaced the dataset used to construct $p(\tau; b)$ with one containing exact solution values. We say that the truncation error is of order ρ at τ_0 if $\mathrm{TE}(r, \tau_0) = \mathcal{O}(r^\rho)$.

3.3. Polynomial time integrators. A *polynomial time integrator* is any time-integration method where each stage and output is computed by evaluating an ODE polynomial formed from an ODE dataset containing the method's inputs, outputs, and stages, along with their corresponding derivatives. Certain polynomial methods can already be found within many classes of existing time integrators, including linear multistep methods, Runge–Kutta methods, and general linear methods. For example, Adams–Bashforth and AM are polynomial methods, as are the Runge–Kutta midpoint and Heun methods, whose coefficients can be derived via polynomial quadrature.

In this work, we view each of these existing methods as a special case of a more general family of parametrized integrators with the ability to scale the interpolation nodes independently of the stepsize. To formally introduce this concept we begin by presenting our notation and by discussing two important families of integrators that emerge from this parametrization.

3.3.1. Parameters and notation. During the timestep from t_n to $t_{n+1} = t_n + h$, a polynomial time integrator accepts q inputs, computes s stages, and produces q outputs. We will denote these quantities as follows:

$$\left.\begin{array}{ll} \text{inputs} & y_j^{[n]} \\ \text{outputs} & y_j^{[n+1]} \end{array}\right\} \; j = 1, \ldots, q, \qquad \text{stages} \quad Y_j, \quad j = 1, \ldots, s.$$

Each input, output, and stage approximates the solution $y(t)$ at a time node. We will express the input times using the variables $t_j^{[n]}$ and the stage nodes using the variables T_j such that

$$y_j^{[n]} \approx y\left(t_j^{[n]}\right), \quad y_j^{[n+1]} \approx y\left(t_j^{[n]} + h\right), \quad \text{and} \quad Y_j \approx y\left(T_j^{[n]}\right).$$

For a classical time-integration method, the input times scale with the stepsize h. For a polynomial method the input times scale with the *node radius* r, which is independent of the stepsize. This additional parameter allows a polynomial method to maintain a specific stepsize while scaling its input times relative to the local smoothness of the solution (see Figure 2).

We express the input times in terms of a node set $\{z_j\}_{j=1}^q$, where $|z_j| \leq 1$. The input times are obtained by scaling the node set by the radius r and translating by the current timestep center:

$$\text{input times} \quad t_j^{[n]} = r z_j + t_n, \quad j = 1, \ldots, q.$$

Similarly, we express the stage times in terms of a node set $\{c_j\}_{j=1}^q$, where the constants c_j may implicitly depend on the parameters r and h:

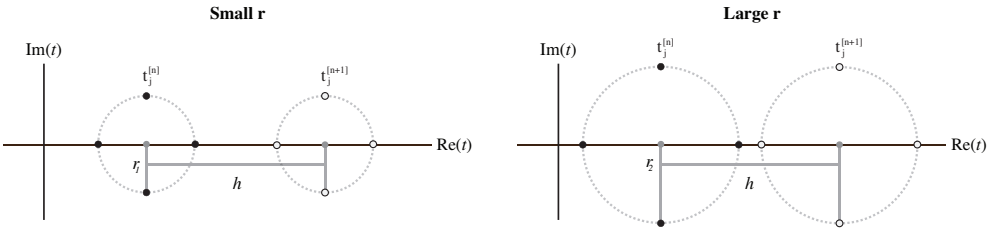


FIG. 2. *Input times $t_j^{[n]}$ (black circles) and output times $t_j^{[n+1]}$ (white circles) for a method whose nodes are equal to the four roots of unity. Input and output times are shown for two different r values where the method stepsize h has been kept constant.*

$$\text{stage times} \quad T_j = rc_j + t_n, \quad j = 1, \ldots, s.$$

Finally, the derivatives for each input and output will be denoted as

$$\begin{aligned} f_j^{[n]} &= F(t_j^{[n]}, y_j^{[n]}), \\ f_j^{[n+1]} &= F(t_j^{[n+1]}, y_j^{[n+1]}), \end{aligned} \qquad j = 1, \ldots, q,$$

while the stage derivatives will be denoted as $F_j = F(T_j, Y_j)$, $j = 1, \ldots, s$.

At times, it is notationally convenient to use vector notation to express the inputs, outputs, and stages. We therefore introduce the input vector $\mathbf{y}^{[n]}$ and the input derivative vector $\mathbf{f}^{[n]}$, where

$$\mathbf{y}^{[n]} = \left[y_1^{[n]}, \ \ldots, \ y_q^{[n]}\right]^{\mathrm{T}} \qquad \text{and} \qquad \mathbf{f}^{[n]} = \left[f_1^{[n]}, \ \ldots, \ f_q^{[n]}\right]^{\mathrm{T}}.$$

We also introduce the similarly defined output vector $\mathbf{y}^{[n+1]}$, output derivative vector $\mathbf{f}^{[n+1]}$, stage vector $\mathbf{Y}$, and stage derivative vector $\mathbf{F}$.

3.3.2. Parametrizing the stepsize: The extrapolation factor α. We will describe polynomial timeintegrators in terms of their ODE polynomials rather than their coefficients. By parametrizing the stepsize h in terms of the node radius r, we obtain natural variables for working with polynomials in local coordinates. We introduce the extrapolation factor α and parametrize the stepsize as

$$h = r\alpha,$$

where the extrapolation factor represents the number of radii r per timestep h. If a polynomial time integrator has complex nodes, then smaller α will require more analyticity in the solution per timestep h. To minimize analyticity requirements, we seek methods with large α, though we will see in subsequent sections that this will not always be possible for reasons of stability and round-off error.

In summary, using the new parametrization, the input times, output times, and stage times for every polynomial method are determined from the following parameters:

$\{z_j\}_{j=1}^q$	nodes	r	node radius
$\{c_j(\alpha)\}_{j=1}^s$	stage nodes	α	extrapolation factor

Since we are only interested in methods that advance the solution at each timestep, we will only consider methods where $\alpha > 0$.

3.4. Polynomial block methods. Block methods [12, 13] generate a set, or block, of q new values at each timestep. They are natural candidates for exploring polynomial time integration since we may use the multivalue input to form high-order polynomial approximations of the differential equation solution. We restrict ourselves to considering block methods of the form

$$\mathbf{y}^{[n+1]} = \mathbf{A}\mathbf{y}^{[n]} + h\mathbf{B}\mathbf{f}^{[n]} + \mathbf{C}\mathbf{y}^{[n+1]} + h\mathbf{D}\mathbf{f}^{[n+1]}, \tag{3.6}$$

where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ are $q \times q$ coefficient matrices. These block methods do not compute any stages or off-step points, and all outputs are computed by taking linear combinations of the inputs, the outputs, and the corresponding derivatives. We distinguish

between six types of block methods based on their *architecture* (parallel or serial) and their degree of *implicitness* (explicit, diagonally implicit, and fully implicit). For parallel diagonally implicit schemes, the q nonlinear systems are independent and may be solved simultaneously, while for parallel explicit schemes, the right-hand-side evaluations may be computed simultaneously. In Table 1 we classify these schemes based on the structure of their coefficient matrices.

TABLE 1
Block methods (3.6) *classified by matrix structure. The abbreviation SLT stands for strictly lower triangular.*

	Explicit	*Diagonally implicit*	*Fully implicit*
Parallel	$\mathbf{C} = \mathbf{D} = \mathbf{0}$	$\mathbf{D}$ diagonal, $\mathbf{C} = \mathbf{0}$	no such methods
Serial	$\mathbf{C}$, $\mathbf{D}$ SLT	$\mathbf{C}$, $\mathbf{D}$ lower triangular	$\mathbf{C}$, $\mathbf{D}$ may be dense

Recall that for polynomial integrators we parameterize the stepsize as $h = r\alpha$, where r scales the input times and α is the number of radii r per timestep. We may express classical block methods using this parametrization by considering integrators of the form

$$\mathbf{y}^{[n+1]} = \mathbf{A}(\alpha)\mathbf{y}^{[n]} + r\mathbf{B}(\alpha)\mathbf{f}^{[n]} + \mathbf{C}(\alpha)\mathbf{y}^{[n+1]} + r\mathbf{D}(\alpha)\mathbf{f}^{[n+1]}. \tag{3.7}$$

The stepsize for a parametrized block method is given by $h = r\alpha$, where the point radius r acts as a scaling factor for the input and output times. A parametrized block method is a polynomial method if the coefficient matrices $\mathbf{A}(\alpha)$, $\mathbf{B}(\alpha)$, $\mathbf{C}(\alpha)$, and $\mathbf{D}(\alpha)$ can be derived from ODE polynomials. However, we will avoid the coefficient formulation (3.7) in favor of a description that explicitly reveals the underlying polynomial approximations.

3.4.1. General form. PBMs compute each output by evaluating an implicit or explicit polynomial approximation of the solution. To construct a PBM, one must choose a set of nodes and a set of ODE polynomials formed from the method's input and output data. Every PBM depends on the parameters

q	number of inputs/outputs	$\{z_j\}_{j=1}^q$	nodes, $z_j \in \mathbb{C}$, $\lvert z_j\rvert \le 1$
r	node radius, $r \ge 0$	$\{b_j\}_{j=1}^q$	expansion points
α	extrapolation factor		

and can be written as

$$y_j^{[n+1]} = p_j(z_j + \alpha;\ b_j), \qquad j = 1, \dots, q, \tag{3.8}$$

where each $p_j(\tau; b)$ is an ODE polynomial over the *ODE dataset*

$$D(r, t_n) = \begin{cases} \text{inputs} : \left\{ \left(z_j,\ y_j^{[n]},\ rf_j^{[n]} \right) \right\}_{j=1}^q, \\ \text{outputs} : \left\{ \left(z_j + \alpha,\ y_j^{[n+1]},\ rf_j^{[n+1]} \right) \right\}_{j=1}^q \end{cases} \tag{3.9}$$

that contains the method's inputs and outputs and their derivatives.

We may classify PBMs based on their architecture and degree of implicitness by specifying the subsets of $D(r, t_n)$ that can be used to form the ODE polynomial. This

classification provides a simple way to construct methods of a particular type. For each type of method listed in Table 1, the polynomial $p_j(\tau; b)$ may use the data

$$\begin{aligned} &\text{inputs} \quad y_k^{[n]},\ f_k^{[n]} \text{ for } k \in \{1, \dots, q\}, \\ &\text{outputs} \quad y_k^{[n+1]},\ f_k^{[n+1]} \text{ for } k \in \mathbb{A}, \end{aligned} \tag{3.10}$$

where the set $\mathbb{A}$ is defined in Table 2.

TABLE 2
For a PBM, the ODE polynomial $p_j(\tau; b)$ may be formed using all input data together with the output data $y_k^{[n+1]}$, $f_k^{[n+1]}$ for $k \in \mathbb{A}$.

	Explicit	*Diagonally implicit*	*Fully implicit*
Parallel	$\mathbb{A} = \emptyset$	$\mathbb{A} = \{j\}$	no such methods
Serial	$\mathbb{A} = \{1, \dots, j-1\}$	$\mathbb{A} = \{1, \dots, j\}$	$\mathbb{A} = \{1, \dots, q\}$

3.5. Linear stability. The linear stability properties of a multivalue method that produces q outputs at each timestep are determined using the Dahlquist test problem $y' = \lambda y$. The timestep iteration for such a method applied to this problem will reduce to

$$\mathbf{y}^{[n+1]} = \mathbf{M}(z)\mathbf{y}^{[n]},$$

where $z = h\lambda$ and $\mathbf{M}(z)$ is a $q \times q$ matrix. The stability region S is the subset of the complex z-plane where $\mathbf{M}(z)$ is power bounded, so that

$$S = \left\{ z \ : \ \sup_{n \in \mathbb{N}} \|\mathbf{M}(z)^n\| < \infty \right\}.$$

In general, the matrix $\mathbf{M}(z)$ will be power bounded if its eigenvalues lie inside the closed unit disk, and any eigenvalues of magnitude one are nondefective.

We can trivially extend linear stability analysis to parameterized integrators. When applied to the Dahlquist test problem, these methods reduce to the iteration

$$\mathbf{y}^{[n+1]} = \mathbf{M}(\zeta, \alpha)\mathbf{y}^{[n]},$$

where $\zeta = r\lambda$ and $\mathbf{M}(\zeta, \alpha)$ is a $q \times q$ matrix. We then define the stability region of the method to be the subset of the complex ζ-plane

$$S(\alpha) = \left\{ \zeta \ : \ \sup_{n \in \mathbb{N}} \|\mathbf{M}(\zeta/\alpha, \alpha)^n\| < \infty \right\}.$$

We take $\zeta \to \zeta/\alpha = z$ so that the stability regions for parametrized methods are scaled relative to the stepsize h instead of the node radius r. This rescaling allows us to overlay the stability regions for parametrized methods directly with those of classical methods.

3.6. Order of accuracy. We can determine the order of accuracy of a polynomial integrator by expressing it as a general linear method, specifying a starting method, and analyzing the resulting algebraic order conditions [6, sect. 53]. Unfortunately, this processes is tedious and the result is dependent on the choice of starting

method. However, we can easily obtain a lower bound on the order of accuracy for polynomial integrators by taking the minimum of the following two quantities: (1) the lowest order of accuracy of our inputs, and (2) the smallest order of the truncation error of the method's ODE polynomials at their evaluation points. This observation leads to the following result.

Remark 3.4. A polynomial time integrator is at least order ρ accurate with respect to the node radius r if the following two conditions are met:
1. The starting method produces inputs that are at least order ρ accurate with respect to r.
2. All of the method's ODE polynomials for computing stages, outputs, and interpolated values have a truncation error of at least order $\rho + 1$ at their evaluation points.

To formally prove order of accuracy [6, sect. 53], we must require that the inputs be order ρ accurate. However, in practice it may be more convenient to compute the starting values using a low-order integrator with a strict error tolerance.

3.6.1. A lower bound for the truncation error of ODE polynomials. The truncation error for any ODE polynomial $p(\tau; b)$ of degree g can be determined by (1) replacing any approximate data values used to form $p(\tau; b)$ with exact values, and (2) Taylor expanding. If the construction strategy for an ODE polynomial is known, then a lower bound on its truncation error is

$$\rho = \min\left(g + 1,\ \lambda_1,\ \ldots,\ \lambda_g\right), \tag{3.11}$$

where the constants λ_j are computed as follows:
1. If the approximate derivative $a_j(b)$ is computed by differentiating a polynomial approximation $h_j(\tau) \approx y(t(\tau))$ of degree d, then

$$\lambda_j = \begin{cases} \infty & \text{if } j = 0 \text{ and } h_j(\tau) \text{ interpolates a solution value at } \tau = b, \\ \infty & \text{if } j = 1 \text{ and } h_j'(\tau) \text{ interpolates a derivative value at } \tau = b, \\ d + 1 & \text{otherwise.} \end{cases}$$

2. If the approximate derivative $a_j(b)$ is computed by differentiating a polynomial approximation $l_j(\tau) \approx y'(t(\tau))$ of degree d, then

$$\lambda_j = \begin{cases} \infty & \text{if } j = 1 \text{ and } l_j(\tau) \text{ interpolates a derivative value at } \tau = b, \\ d + 2 & \text{otherwise.} \end{cases}$$

To derive the bound (3.11), we begin by noticing that the constants λ_j are lower bounds on the truncation error of the approximate derivatives $a_j(b)$. The jth approximate derivative has a truncation error of order λ_j if

$$\left| \tilde{a}_j(b) - r^j y^{(j)}(rb + s) \right| = \mathcal{O}\left(r^{\lambda_j}\right),$$

where $\tilde{a}_j(b)$ was computed using an ODE dataset where any approximate solution values have been replaced with exact ones. The lower bounds for the truncation error orders λ_j are obtained as follows. If the jth derivative of the polynomial approximation for computing $a_j(b)$ interpolates the jth derivative of the solution at $\tau = b$, then the truncation will be infinite since $\tilde{a}_j(b) = y^{(j)}(b)$. If the approximate derivative is computed by differentiating an approximation to $y(t(\tau))$, then the truncation error

order must be greater than or equal to the order of the polynomial plus one. Similarly, if the approximate derivative is computed by differentiating an approximation to $y'(t(\tau))$, then the truncation error order must be greater than or equal to the order of the polynomial plus two since all derivative data is multiplied by a factor of r in local coordinates.

The bound (3.11) can be used to rapidly construct high-order methods by appropriately choosing their ODE polynomials. To construct an ODE polynomial with truncation error ρ, construct all the approximate derivatives $a_j(b)$ by differentiating polynomials approximations of $y(t(\tau))$ that pass through ρ data points, or by differentiating polynomial approximations of $y'(t(\tau))$ that pass through $\rho - 1$ data points. For BDF polynomials (3.5) this is equivalent to choosing an $H(\tau)$ of degree $\rho - 1$. For Adams polynomials (3.4) choose $L_F(\tau)$ of degree $\rho - 2$ and choose b_j to be node values that $L_y(\tau)$ passes through, or choose $L_y(\tau)$ of degree ρ and $L_F(\tau)$ of degree $\rho - 1$.

3.7. Deriving method coefficients. Each stage and output of a polynomial time integrator is computed by evaluating an ODE polynomial at a specific point. This operation amounts to taking a weighted linear combination of solution and derivative data. Each weight depends on the evaluation location of the ODE polynomial and can be computed by solving linear systems. The total number of linear systems is dependent on the number of unique interpolating polynomials used to compute the ODE polynomial's approximate derivatives, and the size of each linear system is given by the order of the interpolating polynomials. The condition number of the resulting linear systems will be heavily influenced by the temporal locations of the data. For certain nodes, such as the roots of unity, the resulting matrices will be perfectly well-conditioned, while for other nodes, such as real-valued nodes, the condition number can grow exponentially.

For fixed time-stepping the coefficients must only be computed once, while for adaptive time-stepping it will be necessary to recompute the weights at each step. In either case, the cost of computing coefficients will be negligible for any reasonably interesting problem since the size of the linear systems will be significantly smaller than the dimension of the initial value problem.

We now describe a procedure for generating the weights for evaluating an ODE polynomial

$$p(\tau; b) = \sum_{j=0}^{g} \frac{a_j(b)(\tau - b)^j}{j!}$$

built from an ODE dataset $D(r, s) = \{(\tau_j, y_j, rf_j)\}_{j=1}^{w}$. We begin by introducing the data vector $\mathbf{d} = [y_1, \ldots, y_w,\ rf_1, \ldots, rf_w]^{\mathrm{T}}$ that contains all the solution and derivative data in $D(r, s)$. Next, we rewrite $p(\tau; b)$ as a weighted linear combination of the data elements, where each weight depends on τ and b, via

$$p(\tau; b) = \mathbf{w}(\tau; b) \cdot \mathbf{d},$$

where $\mathbf{w}(\tau; b) \in \mathbb{C}^{2w}$. The weight vector $\mathbf{w}(\tau; b)$ can be written as

$$\mathbf{w}(\tau; b) = \sum_{j=0}^{g} (\tau - b)^j \mathbf{a}(j; b),$$

where the vectors $\mathbf{a}(j;b)$ no longer depend on τ and satisfy

$$\mathbf{a}(j;b)\cdot\mathbf{d} = \frac{a_j(b)}{j!}, \qquad j = 0,\dots,g.$$

To simplify our discussion for computing the vectors $\mathbf{a}(j;b)$, we introduce the following variables:

$\mathbb{A} = \{\mathbb{A}_i\}_{i=1}^{\gamma}$	A set of size γ containing indices of the solution data used to compute $a_j(b)$.
$\mathbb{B} = \{\mathbb{B}_i\}_{i=1}^{\phi}$	A set of size ϕ containing indices of the derivative data used to compute $a_j(b)$.
$\nu = \gamma + \phi - 1$	The degree of the interpolating polynomial for computing $a(j;b)$ is given by ν.

The approximate derivatives $a_j(b)$ are computed by differentiating interpolating polynomials that pass through solution and derivative values. For notational simplicity, we assume that the temporal nodes of these data values are unique such that $\tau_k \neq \tau_l$ for all $k,\ l \in \mathbb{A}\cup\mathbb{B}$. Though notationally tedious, this algorithm can be easily generalized for polynomials that pass through convex combinations of values at the same time point. However, assuming unique temporal nodes, we may compute $\mathbf{a}(j;b)$ as follows:

1. If $a_j(b)$ is computed by differentiating a polynomial approximation to $y(t(\tau))$, then the polynomial approximation to $y(t(\tau))$ can expanded around $\tau = b$ as

   $$h_j(\tau) = \sum_{k=0}^{\nu} c_k(\tau - b)^k, \qquad \text{where the coefficients } c_k \text{ must satisfy the conditions}$$

   $$\sum_{k=0}^{\nu} c_k(\tau_l - b)^k = y_l \quad \forall l \in \mathbb{A} \qquad \text{and} \qquad \sum_{k=1}^{\nu} k c_k(\tau_l - b)^{k-1} = r f_l \quad \forall l \in \mathbb{B}.$$

   These conditions will lead to a $(\nu+1)\times(\nu+1)$ linear system $\mathbf{H}\mathbf{c} = \mathbf{y}$, where

   $$\left[\begin{array}{ccccc} 1 & \tau_{\mathbb{A}_1} & \tau_{\mathbb{A}_1}^2 & \dots & \tau_{\mathbb{A}_1}^{\nu} \\ \vdots & \vdots & & & \vdots \\ 1 & \tau_{\mathbb{A}_\gamma} & \tau_{\mathbb{A}_\gamma}^2 & \dots & \tau_{\mathbb{A}_\gamma}^{\nu} \\ \hline 0 & 1 & 2\tau_{\mathbb{B}_1} & \dots & \nu\tau_{\mathbb{B}_1}^{\nu-1} \\ \vdots & \vdots & & \vdots & \\ 0 & 1 & 2\tau_{\mathbb{B}_\phi} & \dots & \nu\tau_{\mathbb{B}_\phi}^{\nu-1} \end{array}\right] \left[\begin{array}{c} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_\nu \end{array}\right] = \left[\begin{array}{c} y_{\mathbb{A}_1} \\ \vdots \\ y_{\mathbb{A}_\gamma} \\ \hline r f_{\mathbb{B}_1} \\ \vdots \\ r f_{\mathbb{B}_\phi} \end{array}\right].$$

   If $\mathbf{H}$ is invertible, then the nonzero entries of the vector $\mathbf{a}(j;b)$ are given by

   $$[\mathbf{a}(j;b)]_{\mathbb{A}_k} = \mathbf{H}^{-1}_{j+1,k}, \quad k = 1,\dots,\gamma, \quad \text{and} \quad [\mathbf{a}(j;b)]_{\mathbb{B}_k} = \mathbf{H}^{-1}_{j+1,k+\gamma}, \quad k = 1,\dots,\phi.$$

   If $\mathbf{H}$ is noninvertible, then the $p(\tau;b)$ either is nonunique or does not exist. In either case, we will consider such polynomials not suitable for time-stepping.
2. If $a_j(b)$ is computed by differentiating a polynomial approximation to $y'(t(\tau))$, then $\gamma = 0$, and the polynomial approximation to $y'(t(\tau))$ can be expanded around $\tau = b$ as

   $$l_j(\tau) = \sum_{k=0}^{\nu} c_k(\tau - b)^k, \qquad \text{where} \qquad \sum_{k=0}^{\nu} c_k(\tau_l - b)^k = r f_l \quad \forall l \in \mathbb{B}.$$

These conditions will lead to a $(\nu+1)\times(\nu+1)$ Vandermonde system $\mathbf{Lc}=\mathbf{f}$, where

$$\begin{bmatrix} 1 & \tau_{\mathbb{B}_1} & \dots & \tau_{\mathbb{B}_1}^{v} \\ \vdots & \vdots & & \vdots \\ 1 & \tau_{\mathbb{B}_\gamma} & \dots & \tau_{\mathbb{B}_\gamma}^{v} \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_\nu \end{bmatrix} = \begin{bmatrix} rf_{\mathbb{B}_1} \\ \vdots \\ rf_{\mathbb{B}_\phi} \end{bmatrix}.$$

The nonzero entries of the vector $\mathbf{a}(j,b)$ are given by

$$[\mathbf{a}(j;b)]_{\mathbb{B}_k} = \mathbf{L}^{-1}_{j,k}/j, \qquad k=1,\dots,\phi.$$

4. Constructing PBMs. A PBM with q outputs and nodes $\{z_j\}_{j=1}^q$ can be written as

$$y_j^{[n+1]} = p_j(z_j+\alpha;b_j), \qquad\qquad j=1,\dots,q,$$

where the ODE polynomials $p_j(\tau;b)$ are built from the dataset (3.9). To construct a PBM one must choose a set of ODE polynomials, a set of nodes, and a set of expansion points. In general, these parameters can be chosen in any order. However, selecting one of these parameters first will inform the remaining choices. For example, choosing a particular set of nodes informs the selection of ODE polynomials. We used this strategy in section 2, where we first chose the nodes to be the roots of unity and then built a scheme using Adams polynomials. In this section we employ an alternative construction strategy where we simply choose a set of ODE polynomials without specifying the nodes. This approach generates a family of integrators that are characterized by their ODE polynomials; to obtain a specific method within a family one must select a single set of nodes and endpoints. In the following subsection, we will present several example ODE polynomials for constructing families of block methods and use them to introduce two new schemes that generalize BDF and AM.

4.1. Example Adams PBMs. An Adams PBM is a PBM formed using only the Adams ODE polynomials introduced in subsection 3.2.1. We may express any Adams PBM in integral form as

$$y_j^{[n+1]} = L_y^{[j]}(b_j) + r\int_{b_j}^{z_j+\alpha} L_F^{[j]}(\tau)d\tau, \qquad\qquad j=1,\dots,q.$$

To simplify the presentation of Adams methods, we introduce a compact notation to describe the Lagrange polynomials $L_y^{[j]}(\tau)$ and $L_F^{[j]}(\tau)$. If $L_y^{[j]}(\tau)$ passes through the solution values in an ODE dataset with indices $c_1,\dots,c_\gamma$ and $L_F^{[j]}(\tau)$ passes through the derivative values with indices $d_1,\dots,d_\kappa$, then we will use the following notation:

$$L_y^{[j]} : \left\{y_{c_1},\dots,y_{c_\gamma}\right\}, \qquad\qquad L_F^{[j]} : \left\{f_{d_1},\dots,f_{d_\kappa}\right\}. \tag{4.1}$$

Below we list several possibilities for choosing $L_y^{[j]}(\tau)$ and $L_F^{[j]}(\tau)$. Each of the following polynomials leads to a parallel method with an order of accuracy of at least $q-1$ for any choice of unique nodes $\{z_j\}_{j=1}^q$, endpoints $\{b_j\}_{j=1}^q$, and extrapolation parameter $\alpha>0$.

1. *Parallel explicit*, a block generalization of Adams–Bashforth:

$$L_y^{[j]} : \left[y_1^{[n]},\ y_2^{[n]},\ \dots,\ y_q^{[n]}\right], \qquad L_F^{[j]} : \left[f_1^{[n]},\ f_2^{[n]},\ \dots,\ f_q^{[n]}\right]. \tag{4.2}$$

2. *Parallel diagonally implicit*, a block generalization of AM: If the jth output node overlaps with any input node, i.e., there exists an integer $k(j) \in \{1, \dots, q\}$ such that $z_{k(j)} = z_j + \alpha$, then choose the polynomials

$$L_y^{[j]} : \left[y_1^{[n]},\ y_2^{[n]},\ \dots,\ y_q^{[n]} \right], \qquad L_F^{[j]} : \left[\phi_{j,1}^{[n]},\ \phi_{j,2}^{[n]},\ \dots,\ \phi_{j,q}^{[n]} \right], \qquad \phi_{j,l}^{[n]} = \begin{cases} f_l^{[n+1]} & \text{if } l = k(j), \\ f_l^{[n]} & l \neq k(j). \end{cases} \tag{4.3}$$

If there is no overlap, then choose the polynomials

$$L_y^{[j]} : \left[y_1^{[n]},\ y_2^{[n]},\ \dots,\ y_q^{[n]} \right], \qquad L_F^{[j]} : \left[f_1^{[n]},\ f_2^{[n]},\ \dots,\ f_q^{[n]},\ f_j^{[n+1]} \right]. \tag{4.4}$$

4.2. Families of BDF PBMs. A BDF PBM is a PBM formed using only the BDF ODE polynomials introduced in subsection 3.2.1. We may write these methods simply as

$$y_j^{[n+1]} = H_y^{[j]}(z_j + \alpha), \qquad j = 1, \dots, q.$$

If $H_y^{[j]}(\tau)$ interpolates the solution values in an ODE dataset with indices $c_1, \dots, c_\gamma$ and $\frac{d}{dt} H_y^{[j]}(\tau)$ interpolates a derivative value with index κ, then we will use the following notation:

$$H_y^{[j]} : \left\{ y_{c_1},\ \dots\ ,\ y_{c_\gamma},\ f_\kappa \right\}. \tag{4.5}$$

Below we list several possibilities for choosing $H_y^{[j]}(\tau)$ for constructing parallel methods. If each of the polynomials $H_y^{[j]}(\tau)$ can be constructed for $j = 1, \dots, q$, then the formulas will produce methods with order of accuracy q. However, for certain node sets $\{z_j\}_{j=1}^q$ and α values it will not be possible to construct the polynomial $H_y^{[j]}(\tau)$ since the corresponding linear systems will be singular.

1. *Parallel explicit*, a new class of explicit BDF methods:

$$H_y^{[j]} : \left[y_1^{[n]},\ y_2^{[n]},\ \dots,\ y_q^{[n]},\ f_j^{[n]} \right]. \tag{4.6}$$

2. *Parallel diagonally implicit*, a block generalization of classical BDF:

$$H_y^{[j]} : \left[y_1^{[n]},\ y_2^{[n]},\ \dots,\ y_q^{[n]},\ f_j^{[n+1]} \right]. \tag{4.7}$$

4.3. Classical linear multistep methods as special cases of PBMs. The *ODE interpolating polynomials* described in subsections 4.1 and 4.2 will produce classical linear multistep methods for special nodes, endpoints, and extrapolation parameters. For example, if we choose

$$z_j = -1 + 2\left(\tfrac{j-1}{q-1}\right), \qquad b_j = \begin{cases} z_{j+1}, & j < q, \\ z_q, & j = q, \end{cases} \qquad \text{and} \qquad \alpha = \tfrac{2}{q-1},$$

then (4.2) produces the classical Adams–Bashforth method of order q, (4.3) and (4.4) produce the classical AM method of order $q+1$, and (4.7) produces the BDF of order q.

4.4. New polynomial block methods with complex nodes. We now introduce two new PBMs that are inspired by classical BDF and AM. These methods use a node set consisting of q equispaced points on the imaginary interval $[-i,\ i]$, given by

$$z_j = -i + 2i(j-1)/(q-1), \qquad j = 1, \ldots, q. \tag{4.8}$$

These nodes are merely a rotation of those used to obtain the classical BDF and AM methods. We may describe these new methods as follows:

1. *BBDF*: A parallel, diagonally implicit block BDF method of order q with ODE polynomials (4.7), node set (4.8), and $q > 1$.
2. *BAM*: A parallel, diagonally implicit block AM method of order $q+1$ with ODE polynomials (4.4), node set (4.8), expansion points $b_j = z_j$, and $q > 1$.

When q is even, the BBDF and BAM methods have no real-valued nodes. We must therefore compute an additional output at the local coordinate $\tau = \alpha$ during the final timestep, or at any timestep where we wish to obtain the solution at a real time point. This can be accomplished by forming and evaluating a new ODE polynomial $p_{\text{out}}(\tau; b)$ so that

$$y_{\text{out}}^{[n+1]} = p_{\text{out}}(\alpha; b_{\text{out}}).$$

Possible choices for the output polynomials for BBDF and BAM are

1. $BBDF_{\text{out}}$ $\qquad H_F^{[j]} : \left[y_1^{[\text{n}]}, \ldots, y_q^{[\text{n}]},\ f_{\text{out}}^{[\text{n+1}]}\right],$

2. BAM_{out} $\qquad \begin{cases} L_y^{[j]} : \left[y_1^{[\text{n}]}, \ldots, y_q^{[\text{n}]}\right], \quad L_F^{[j]} : \left[f_1^{[n]}, \ldots,\ f_q^{[n]}, f_{\text{out}}^{[n+1]}\right], \\ b_{\text{out}} = z_{q/2}. \end{cases}$

Remarks regarding complex arithmetic. By choosing complex-valued nodes we must now store the solution at complex times and solve complex-valued nonlinear systems. In general this leads to a doubling in storage costs and a doubling in computational complexity. For reference, in Table 3 we highlight the slowdown factor caused by switching from real-valued to complex-valued arithmetic in MATLAB.

TABLE 3

Quantifying the slowdown factor caused by complex arithmetic in MATLAB for scalar multiplication, scalar division, matrix-vector multiplication, and solving linear systems. Random matrices of dimension 100 *were used to generate the last two entries in the table.*

Operation	$a * b$	a/b	$\mathbf{Ax}$	$\mathbf{Ax} = \mathbf{b}$
Slowdown factor	1.4x	1.9x	2.0x	2.2x

During each timestep a polynomial method with complex nodes may require any of the following operations:

- Evaluation of the right-hand-side $F(t, y(t))$ at complex times.
- Computation of linear combinations of complex-valued solution vectors where the weights will also be complex-valued.
- Solution of complex-valued nonlinear systems with complex-valued initial conditions.

For certain methods, including BAM and BBDF, the total storage cost will be equivalent to that of a BDF or Adams–Moulton method of identical order. This is

because the nodes $\{z_j\}_{j=1}^q$ and endpoints $\{b_j\}_{j=1}^q$ for BBDF and BAM are symmetric with respect to the real line. If the underlying ODE is real-valued then, the Schwarz reflection principle implies that the solution $y(t)$ and its derivative $F(t, y(t))$ satisfy the relations

$$y(z^*) = y(z)^* \qquad \text{and} \qquad F(z^*, y(z^*)) = F(z, y(z))^*,$$

where * denotes complex conjugate. For BBDF and BAM methods, this implies that we only need to compute the first $\lceil q/2 \rceil$ outputs and function evaluations, since it follows that

$$y_j^{[n+1]} = \left(y_{q-j+1}^{[n+1]}\right)^* \quad \text{and} \quad f_j^{[n+1]} = \left(f_{q-j+1}^{[n+1]}\right)^* \quad \text{for} \quad j = 1, \ldots, \lfloor q/2 \rfloor.$$

This simplification reduces the storage requirements of the method by half and eliminates redundant computation, but it does not reduce the parallel time of the method since each of the q systems can be solved independently.

Finally, when implementing these integrators on real-valued problems, it is important to set to zero any imaginary rounding errors at each timestep from any outputs with real-valued temporal nodes. If the imaginary rounding errors are not removed, then we find that the method may become unstable.

4.4.1. Linear stability results for BBDF and BAM schemes. The BBDF and BAM methods presented in section 4.4 are block generalizations of BDF and AM. In this section we will compare the stability properties of each of these six schemes with a focus on $A(\theta)$ stability for BDF methods and negative stability intervals for Adams methods. The two properties are defined as follows:

1. A method is $A(\theta)$ stable if its stability region

$$S \supset \{z : |\arg(-z)| < \theta,\ z \neq 0\}.$$

2. A method has a negative stability interval of $[-\beta, 0]$ if its stability region

$$S \supset [-\beta, 0] \text{ for } \beta > 0.$$

For classical BDF methods of orders one through six, $A(\theta)$ stability decreases monotonically, and methods of order greater than six are no longer root stable (see Table 4). On the other hand, classical AM methods have bounded linear stability regions that contract rapidly with order and are often smaller than those of explicit methods (see Table 5).

BBDF and BAM methods improve upon these limitations, leading to high-order methods with more favorable stability properties. In comparison to BDF, BBDF schemes satisfy the root condition past order six and provide improved $A(\theta)$ stability especially for small α. In Table 4 we present $A(\theta)$ stability for BBDF, and in Figure 3 we show plots of BBDF stability regions and those of the classical BDF methods.

Like AM methods, BAM methods have bounded stability regions. However, they possess significantly larger linear stability regions than their classical counterparts, rendering them more useful for mildly stiff equations. One drawback is that they are unstable along the imaginary axis. In Table 5 we present real stability intervals for BAM schemes and in Figure 4 we present plots of BAM stability regions overlaid with those of the classical AM methods.

The stability gains for the BBDF and BAM methods are possible because of the extrapolation parameter α. As a general rule we find that smaller α values lead to improved stability for both methods. However, recall that choosing a smaller α requires a greater amount of analyticity relative to the timestep $h = r\alpha$.

TABLE 4

$A(\theta)$ stability for classical BDF and BBDF from section 4.4. The table lists θ values for methods of orders two through eight and for BBDF methods with α values of 1, 1/2, 1/4, and 1/8. All angles are rounded to two decimals. Empty positions indicate the underlying method is not root stable.

Order	2	3	4	5	6	7	8
BDF	90°	86.03°	73.35°	51.84°	17.84°		
$\text{BBDF}_{\alpha=1}$	90.00°	89.54°	88.51°	87.58°	86.89°		
$\text{BBDF}_{\alpha=\frac{1}{2}}$	90.00°	89.88°	89.32°	88.51°	87.72°	87.05°	83.58°
$\text{BBDF}_{\alpha=\frac{1}{4}}$	90.00°	89.99°	89.90°	89.68°	89.31°	88.83°	88.33°
$\text{BBDF}_{\alpha=\frac{1}{8}}$	90.00°	89.99°	89.99°	89.98°	89.94°	89.86°	89.75°

TABLE 5

Negative stability intervals β for AM and BAM schemes from section 4.4. The table lists β values for methods of orders two through eight and for BAM methods with α values of 1, 1/2, 1/4, and 1/8. All values are rounded to two decimals.

Order	3	4	5	6	7	8
AM	6.00	3.00	1.84	1.18	0.77	0.49
$\text{BAM}_{\alpha=1}$	58.01	11.66	7.24	5.68	4.81	4.23
$\text{BAM}_{\alpha=\frac{1}{2}}$	202.01	29.66	14.34	9.29	7.21	5.90
$\text{BAM}_{\alpha=\frac{1}{4}}$	778.01	101.67	42.77	23.60	15.94	11.88
$\text{BAM}_{\alpha=\frac{1}{8}}$	3082.01	389.67	156.55	81.17	51.19	35.31

5. Numerical experiments. In this section, we present numerical experiments to demonstrate the improved stability properties of BBDF and BAM from section 4.4 in comparison to classical BDF and AM. We compare these implicit methods on two stiff partial differential equations and present plots of absolute error versus stepsize and absolute error versus computational time where errors are measured using the L^∞ norm. For each equation, we compute the reference solutions using the MATLAB ode15s integrator with a tolerance of 1e-14. Inputs for all multivalue methods are computed at the initial timestep using the exponential integrator EPIRK43a [17]. For implicit methods, we solve all nonlinear systems using Newton's method with an exact Jacobian and the MATLAB backslash to solve the underlying linear systems.

All the results presented in this paper have been run on a six-core Intel i7-8700 CPU running at 3.20GHz. The BDF and AM methods were run using a single thread, while the BBDF and BAM methods where run using one thread per nonlinear solve and an additional master thread to combine the results and advance the timestep. All parallelization was done using the MATLAB parallel toolkit which utilizes MPI. The MATLAB implementation used to generate these results can be downloaded from [7].

5.1. Test problems. We conduct our numerical experiments using two stiff partial differential equations. For each equation, we discretize in space using standard second-order finite differences. Below we describe the equations, their initial conditions, and the corresponding numerical parameters.

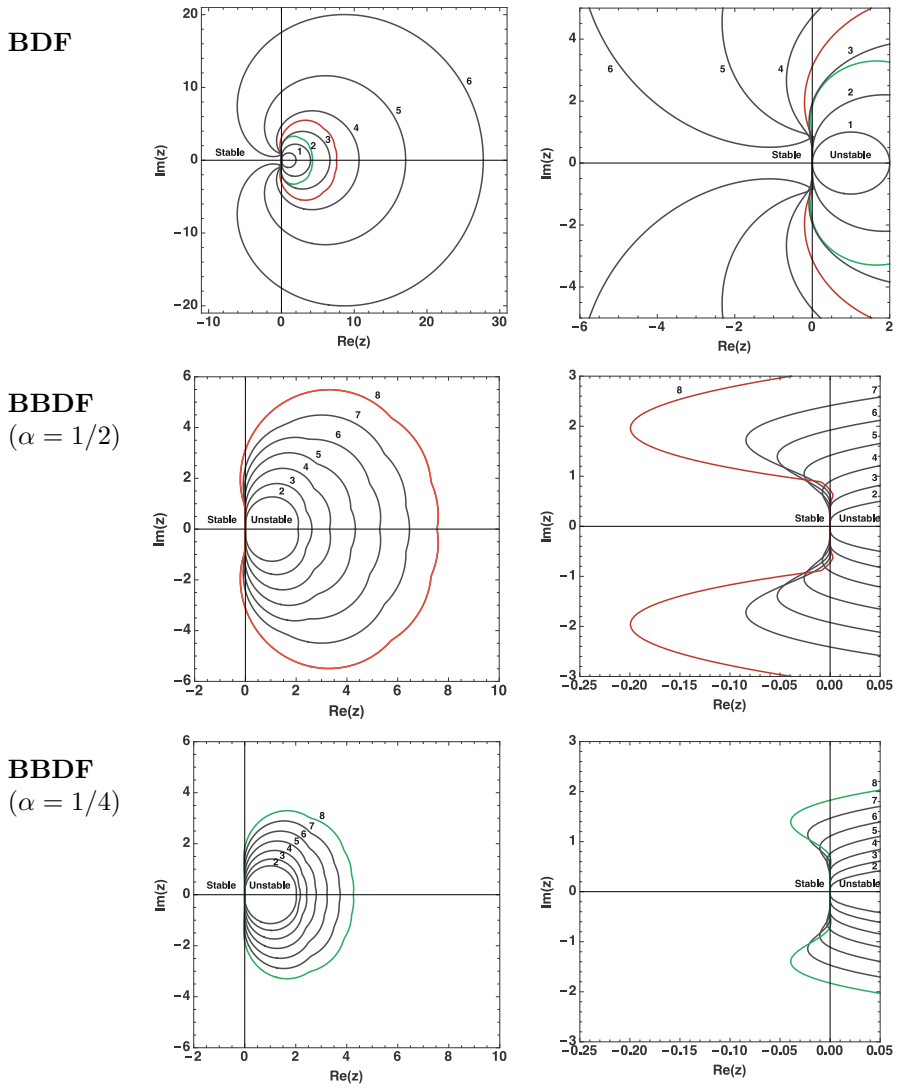


FIG. 3. *Stability regions for BDF methods of orders two through six and BBDF methods of orders two through eight with rotated equispaced nodes. The diagrams on the right are magnified to show the stability region near the imaginary axis. The stability boundaries for eighth-order BBDF with $\alpha = 1/2$ and $\alpha = 1/4$ are respectively labeled in red and green. Decreasing the extrapolation parameter α consistently improves $A(\theta)$ stability for BBDF schemes.*

1. We consider the one-dimensional viscous Burgers' equation with homogeneous boundary conditions [19],

$$
\begin{aligned}
&u_t = \nu u_{xx} - u u_x, \\
&u(x, t=0) = \left(\sin(3\pi x)\right)^2 (1-x)^{3/2}, \\
&x \in [0,1], \quad t \in [0,1],
\end{aligned}
\tag{5.1}
$$

where we take $\nu = 3 \times 10^{-4}$. We discretize in space using 2000 gridpoints and test all time integrators using 15 different stepsizes logarithmically spaced between 5×10^{-3} and 5×10^{-4}. Equation (5.2) evolves rapidly and the timesteps were

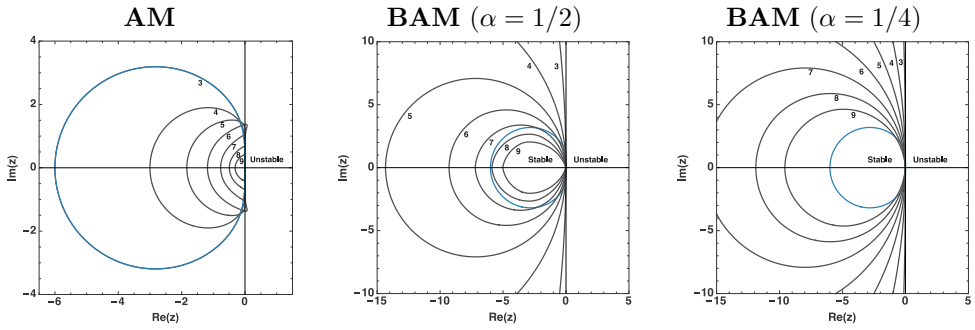


FIG. 4. *Stability regions for AM and BAM of orders three through nine. The stability boundary for third-order AM appears as a blue contour in all figures. Block methods have significantly larger stability regions that do not enclose any portion of the imaginary axis. Smaller α leads to larger and more circular stability regions.*

chosen based on accuracy and not to guarantee sufficient analyticity off the real timeline. As a point of comparison explicit fourth-order Runge–Kutta is only stable for timesteps smaller than 5.8×10^{-4}.

2. We consider the two-dimensional advection–diffusion–reaction (ADR) equation with homogeneous Neumann boundary conditions [17],

$$\begin{aligned} u_t &= \epsilon\,(u_{xx} + u_{yy}) + \delta\,(u_x + u_y) + \gamma u(u - 1/2)(1 - u), \\ u(x, t = 0) &= 256(xy(1 - x)(1 - y))^2 + 0.3 \\ x, y &\in [0, 1], \quad t \in [0, 0.05], \end{aligned} \tag{5.2}$$

where we take $\epsilon = 1/100$, $\delta = -10$ and $\gamma = 100$. We discretize in space using a 400×400 point grid and test all time integrators using 15 different stepsizes logarithmically spaced between 1×10^{-3} and 1×10^{-4}. The ADR equation (5.2) evolves rapidly and the timesteps were chosen based on accuracy and not to guarantee sufficient analyticity off the real timeline. As a point of comparison, explicit fourth-order Runge–Kutta is stable only for timesteps smaller than 2.2×10^{-4}.

5.2. Results for diagonally implicit polynomial block methods. We compare the BBDF and BAM methods from section 4.4 with an extrapolation parameter of $\alpha = 1/2$, against BDF and AM. We also include eighth-order BBDF and BAM methods with $\alpha = 1/4$ to demonstrate the effects of rounding errors on high-order methods. In Figures 5 and 6 we present error versus stepsize and error versus running time diagrams for Burgers' equation and the ADR equation. We draw the following conclusions from our numerical experiments.

1. *Stability and accuracy.* High-order BDF and AM methods were either completely unstable or only stable at smaller stepsizes. BBDF methods were stable at all orders and BAM methods showed improved stability over AM methods of equivalent order. All four classes of methods demonstrated or exceeded their expected orders of accuracy at small stepsizes. When solving the viscous Burgers' equation, BBDF and BAM did not converge at the coarsest stepsizes due to insufficient analyticity of the solution.

We find that AM and BAM methods have superior error properties compared to BDF and BBDF, making them more efficient so long as they remain stable. Rounding errors presented significant problems for both BBDF8, which was unable to achieve an

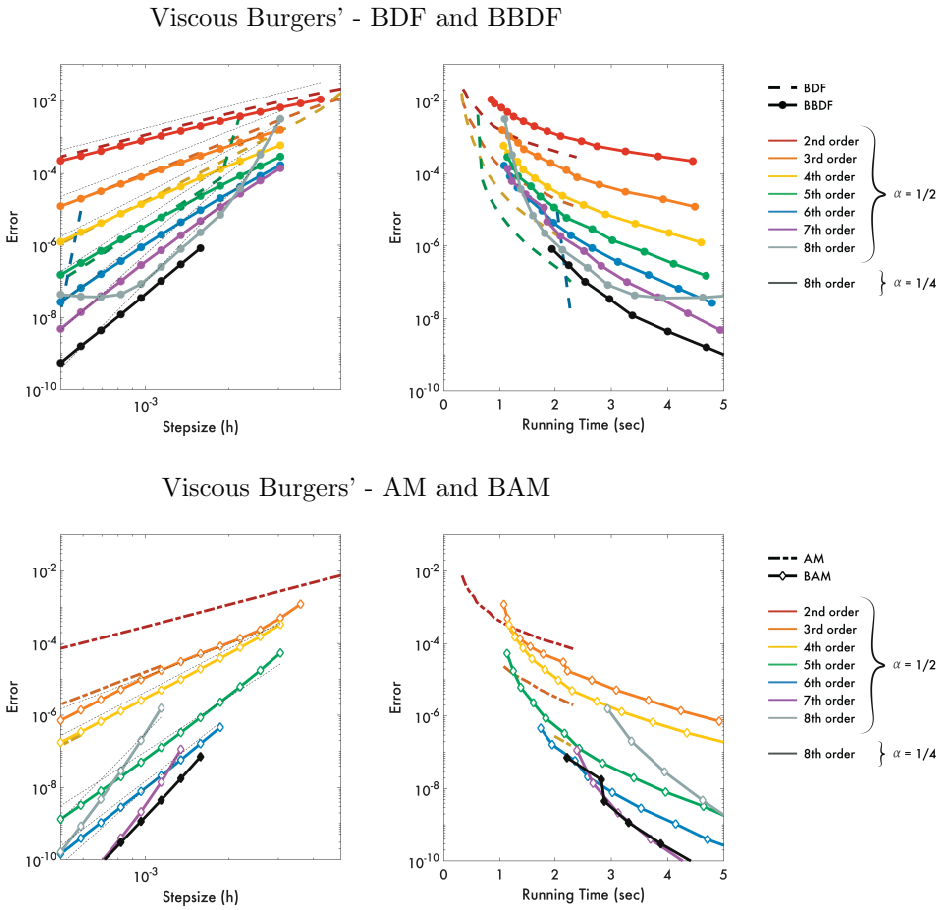


FIG. 5. *Error versus stepsize and error versus running time diagrams for the viscous Burgers' equation. We present results for BDF and BBDF in the top two plots and results for BAM and AM in the bottom two Plots. All BAM and BBDF methods of orders two through seven are run using* $\alpha = 1/2$. *For eighth-order methods we also show results for BAM and BBDF methods with* $\alpha = 1/4$, *to show that reducing* α *ameliorates sensitivity to rounding errors. The seven thinly dashed gray lines on the error versus stepsize plots correspond to second- through eighth-order convergence.*

error below 1e-7, and BAM8, which performed worse than the BAM7 on the Burgers' equation. However, we can resolve this issue by choosing a smaller $\alpha = 1/4$ for these eighth-order schemes.

2. *Efficiency.* Overall, BBDF and BAM methods are comparable in efficiency to BDF and AM schemes. In short, the primary reason to consider BBDF and BAM over BDF or AM schemes is their improved stability at high orders of accuracy. For large stepsizes, high-order BDF schemes are unstable for even mildly dispersive PDEs, and AM methods are not appropriate for stiff equations. In contrast, high-order BBDF methods can be applied to a wider range of problems, and BAM methods will outperform BBDF on dissipative PDEs that are not too stiff. For example, by increasing the number of spatial gridpoints in either of our test problems, we can cause BDF methods of orders greater than 2 to become unstable long before any of their BBDF counterparts.

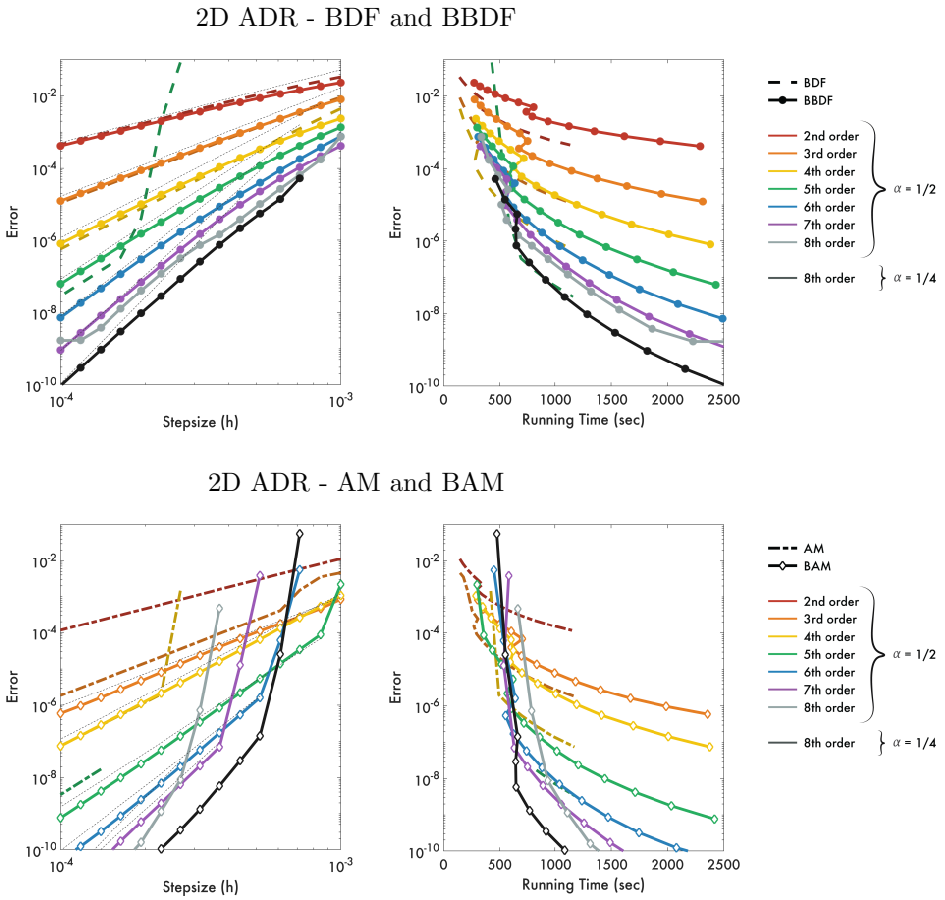


FIG. 6. *Error versus stepsize and error versus running time diagrams for the two-dimensional ADR equation. We present results for BDF and BBDF in the top two plots and results for BAM and AM in the bottom two plots. All BAM and BBDF methods are run using* $\alpha = 1/2$. *The seven thinly dashed gray lines on the error versus stepsize plots correspond to second- through eighth-order convergence. For eighth-order methods we also plot BAM and BBDF methods with* $\alpha = 1/4$ *to show that reducing* α *ameliorates sensitivity to rounding errors.*

In general, the efficiency of BBDF and BAM will be dependent on the computer architecture, the dimension of the initial value problem, and the computational cost of solving the nonlinear systems. For real-valued problems, BBDF and BAM methods of order q require $\frac{q}{2}$ complex nonlinear solves per timestep, while BDF and AM methods require only a single real-valued nonlinear solve. If implemented in serial, BBDF and BAM of order q will cost roughly q times more per timestep than any BDF or AM method. If parallelized, then each of the nonlinear systems can be solved simultaneously at each timestep. If these integrators are run on a distributed memory system, then it will be necessary to communicate all the new solution values between the nodes after each timestep. If the communication cost is high, then optimal parallel efficiency will not be obtained. On a shared memory system communication is not necessary, but cache misses may degrade parallel performance.

Our parallel implementation of BAM and BBDF were created using the MATLAB *spmd* statements, which utilize MPI. Each MATLAB worker has its own memory and

communication is necessary at each timestep. For the ADR equation, the Newton solves require inverting a sparse, banded matrix of dimension 400^2 and the cost of communicating solution vectors is negligible compared to that of solving the nonlinear systems. For Burgers, the linear systems are tridiagonal with dimension 4000 and the cost of inversion is very low; hence, the effects of communication are clearly visible in our efficiency diagrams (this is evidenced by the fact that curves for BBDF and BAM are shifted more to the right than in the efficiency diagrams for the ADR equation).

Finally, there is also the effect of the complex arithmetic. In our implementation, a single step with a BBDF or BAM takes roughly twice as long as a single step with BDF or AM. The additional cost is due to the complex-valued linear solves at each Newton iteration. We use MATLAB backslash (mldivide) to solve linear systems; however, we expect that for large problems one can match the performance of BDF of AM by using a linear solver that allows for additional parallelization. For example, one may rewrite the complex linear systems into a real system of twice the dimension and apply GMRES where the larger matrix multiplications have been parallelized to offset the additional cost.

6. Conclusions. In this paper, we introduced a methodology for constructing time integrators that combines ideas from approximation theory and complex analysis. Our approach eliminates the complexity of order conditions enabling simplified construction of methods with a specific architecture (parallel or serial), degree of implicitness (explicit, diagonally implicit, fully implicit), and desired order of accuracy. Thus far, this framework has enabled us to derive new classes of high-order block methods with improved linear stability regions compared to their classical counterparts. We illustrated the utility of our new approach by constructing generalized BDF and AM integrators. The generality of this polynomial framework raises many interesting theoretical and practical questions about the properties and utility of this approach that we plan to address in our future work. In particular, in a follow-up publication we will construct parametrized general linear methods and pursue the development of efficient adaptive strategies for varying the extrapolation parameter α at each timestep.

Acknowledgments. The authors would like to thank Randall J. LeVeque for many useful discussions spanning the entire course of this project and for his comments on this work. We would also like to thank John C. Butcher for his many helpful discussions about general linear methods and B-Series.

REFERENCES

[1] M. J. Ablowitz and A. S. Fokas, *Complex Variables: Introduction and Applications*, Cambridge University Press, New York, 2003.
[2] A. P. Austin, P. Kravanja, and L. N. Trefethen, *Numerical algorithms based on analytic function values at roots of unity*, SIAM J. Numer. Anal., 52 (2014), pp. 1795–1821.
[3] F. Bornemann, *Accuracy and stability of computing high-order derivatives of analytic functions by Cauchy integrals*, Found. Comput. Math., 11 (2011), pp. 1–63.
[4] F. Bornemann and G. Wechslberger, *Optimal contours for high-order derivatives*, IMA J. Numer. Anal., 33 (2013), pp. 403–412.
[5] J. P. Boyd, *Chebyshev and Fourier Spectral Methods*, Dover, New York, 2013.
[6] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*, John Wiley & Sons, New York, 2016.
[7] T. Buvoli, *Codebase for "Constructing Time Integrators Using Interpolating Polynomials,"* https://doi.org/10.5281/zenodo.3031860 (2019).
[8] G. F. Corliss, *Integrating ODEs in the complex plane—pole vaulting*, Math. Comp., 35 (1980), pp. 1181–1189.

[9] B. Fornberg, *Numerical differentiation of analytic functions*, ACM Trans. Math. Software, 7 (1981), pp. 512–526.
[10] B. Fornberg, *A Practical Guide to Pseudospectral Methods*, Vol. 1, Cambridge University Press, New York, 1998.
[11] B. Fornberg and J. A. C. Weideman, *A numerical methodology for the Painlevé equations*, J. Comput. Phys., 230 (2011), pp. 5957–5973.
[12] M. J. Gander, 50 *years of time parallel time integration*, in Multiple Shooting and Time Domain Decomposition Methods, Springer, New York, 2015, pp. 69–113.
[13] C. W. Gear, *Parallel methods for ordinary differential equations*, Calcolo, 25 (1988), pp. 1–20.
[14] E. Hansen and A. Ostermann, *High order splitting methods for analytic semigroups exist*, BIT, 49 (2009), pp. 527–542.
[15] J. N. Lyness and C. B. Moler, *Numerical differentiation of analytic functions*, SIAM J. Numer. Anal., 4 (1967), pp. 202–210.
[16] T. Orendt, J. Richter-Gebert, and M. Schmid, *Geometry of Numerical Complex Time Integration*, preprint, arXiv:0903.1585, 2009.
[17] G. Rainwater and M. Tokman, *A new approach to constructing efficient stiffly accurate EPIRK methods*, J. Comput. Phys., 323 (2016), pp. 283–309.
[18] J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, 2nd ed., SIAM, Philadelphia, 2004.
[19] M. Tokman, *Efficient integration of large stiff systems of ODEs with exponential propagation iterative (EPI) methods*, J. Comput. Phys., 213 (2006), pp. 748–776.
[20] L. N. Trefethen, *Spectral Methods in MATLAB*, Software Environ. Tools 10, SIAM, Philadelphia, 2000.
[21] L. N. Trefethen, *Approximation Theory and Approximation Practice*, SIAM, Philadelphia, 2013.
[22] L. N. Trefethen and J. Weideman, *The exponentially convergent trapezoidal rule*, SIAM Rev., 56 (2014), pp. 385–458.