An Adaptive, Highly Accurate and Efficient, Parker-Sochacki Algorithm for Numerical Solutions to Initial Value Ordinary Differential Equation Systems

Jenna Guenther, James Madison University and Morgan Wolf, James Madison University
Advisors: Parker-Sochacki Group, Lead: Dr. Paul Warne, James Madison University

Abstract.
The Parker-Sochacki Method (PSM) allows the numerical approximation of solutions to a polynomial initial value ordinary differential equation or system (IVODE) using an algebraic power series method. PSM is equivalent to a modified Picard iteration and provides an efficient, recursive computation of the coefficients of the Taylor polynomial at each step. To date, PSM has largely concentrated on fixed step methods. We develop and test an adaptive stepping scheme that, for many IVODEs, enhances the accuracy and efficiency of PSM. PSM Adaptive (PSMA) is compared to its fixed step counterpart and to standard Runge-Kutta (RK) foundation algorithms using three example IVODEs. In comparison, PSMA is shown to be competitive, often outperforming these methods in terms of accuracy, number of steps, and execution time. A library of functions is also presented that allows access to PSM techniques for many non-polynomial IVODEs without having to first rewrite these in the necessary polynomial form, making PSM a more practical tool.

1. Introduction. Initial value ordinary differential equation systems (IVODEs) lie at the core of many models that characterize our universe. Since explicit closed form analytic solutions to many IVODEs are not feasible, simulations modeled by IVODEs frequently depend on well-developed numerical methods. Such methods regularly have a theoretical connection to Taylor series in approximating solutions. With simple numerical methods, for example, Clairaut, Lalande, and Lepaute were able to make significant predictions in celestial mechanics, including the arrival of Halley's Comet, as early as the 1700s [15]. In the 1800s, John Crouch Adams implemented methods for solving IVODEs using naive predictor-corrector schemes in modeling planetary motion. Sir George H. Darwin of Cambridge later incorporated elementary adaptive stepping [15].

Classic Taylor methods that approximate solutions of IVODEs by continually computing increasingly higher-order derivatives about a point have been used as early as the 1700s [15]. Unfortunately, a classic Taylor approach often requires complicated calculations of extensive symbolic derivatives, and quickly becomes intractable when transcendental and other nonlinear functions are introduced. The numerical Runge-Kutta (RK) algorithms were developed in the early 20^{th} century [15]. In these algorithms, approximations are cleverly determined using nested function evaluations in lieu of calculating successive derivatives. RK methods, specifically of fourth order, are a standard in approximating solutions to IVODEs.

Early RK algorithms were fixed step methods. These were later developed into adaptive algorithms, which are generally more efficient. Adaptive methods frequently allow many fewer steps while retaining similar or better accuracy, in comparison to their fixed step counterparts. In a 1969 report to NASA, Fehlberg first developed what he called "step size control" with low order RK formulas [14]. He derived the overlapping coefficients for the modern day Runge-Kutta-Fehlberg (RKF) method and based the step size in part on a comparison of the local truncation errors of RK methods of orders 4 and 5 [14]. In 1980, Dormand and Prince established an adaptive method that serves as the foundation of several built-in IVODE

solvers, including MATLAB's ODE45 package. Dormand-Prince (DP) has been shown to be more accurate and efficient than RKF for several IVODEs [19]. Given the above, these two standard RK adaptive order 4-5 methods, RKF and DP, are of comparative interest in this work. The most sophisticated adaptive methods are able to simultaneously control total error and local truncation error to satisfy specified tolerances.

In the late 1980s, Parker and Sochacki discovered a method for approximating the solutions to polynomial differential equations based on a modified Picard iteration scheme referred to as the Parker-Sochacki Method. [29]. The early framework allowed theoretic machinery to be applied to ordinary, partial, and integral differential equations [10, 36, 31, 30]. Although able to handle the nonlinear and transcendental functions that cause difficulty for classic Taylor methods, the computation of successive terms of the Taylor polynomials through this modified Picard approach was computationally expensive.

In 2002, [40] derived an algebraic power series method (PSM) which is equivalent to the Picard approach for polynomial IVODEs, allowing for an efficient recursive computation of the coefficients of the Taylor series at each step. PSM generates the Taylor coefficients of the solution efficiently without relying on symbolic calculation of derivatives, making it attractive when compared to standard Taylor methods. Also, higher (or lower) degree approximations only require a reassignment of a parameter in a program, giving the user "on the fly" step-wise control over the order of the algorithm. In addition, PSM provides a Taylor polynomial approximation to the solution across the entire time step rather than at just a single value, a potential advantage over RK strategies. Aside from machine round-off error, PSM can *a-priori* guarantee that, at any given step, the error of the approximation will remain less than a designated desired error tolerance [40]. For a brief overview of PSM, see Section 2 and [9, 10].

Over the last decade, application and interest in PSM for problems of technological and scientific importance has increased dramatically, including techniques for trajectory propagation. For example, PSM was used as the primary solver in [38] for gathering simulation data in testing the times of firing events in neurons and scheduling algorithms used for synaptic event delivery. See [31, 30, 1, 2, 4, 5, 6, 11, 13, 17, 20, 21, 23, 26, 27, 32, 33, 35, 37, 39] for a sample of recent studies that report the advantages (and some disadvantages) of PSM. PSM and work from the Automatic Differentiation (AD) community have developed nearly simultaneously [9, 16, 25], and there has been a number of benefits in collaborations between these two communities, including strategies in this work. Both [12] and [7] discuss the early AD foundation of interval analysis and generating the coefficients of a Taylor polynomial via successive derivatives and recurrence relations, which have clear connections to PSM. Interest in PSM continues to grow, largely thanks to the cogent, robust approach to the n-body problem detailed in [31, 30].

To date, development of PSM for IVODEs has focused largely on fixed step methods [36, 10, 29]. A foundation for an adaptive stepping PSM algorithm similar to RKF and DP could be broadly used to numerically solve a wide class of IVODEs with significant gains. Like RKF and DP, our adaptive PSM algorithm (PSMA) is an explicit one-step method that could represent an engine or base algorithm for future sophisticated IVODE solvers. As work from the AD community developed in parallel with PSM, [18] provided a modern adaptive implementation of AD in ANSI C on Linux that allowed for adaptation in both order and step size. The step size there was similarly derived from an asymptotic error estimate, as

done here. In our PSM work, we demonstrate some greater suitability for stiff problems and a broader library that includes transcendental functions.

To exploit and advance PSM, a major focus of this research is the development of both the theoretical framework for PSMA and the process for choosing step sizes, which follows in Sections 2 and 3. PSM and PSMA are ripe for competitive comparisons with results from RK and DP algorithms. We thus extend our work and apply the adaptive theory on several examples to compare PSM against the standard RK algorithms, particularly in terms of the effect of higher order approximations. Highlighted examples in Section 4 include a singular problem, an IVODE used in simulations that model missile flight trajectories with two degrees of freedom (DoF), and a classic stiff problem used to model flame propagation. All codes were implemented and executed in a Matlab programming environment on one Dell Latitude 3450 computer.

A second important contribution is presented in Section 5. Early PSM methods required the user to first rewrite the IVODEs in a polynomial form. To avoid this step, a versatile library of succinct PSM specific functions, which make PSM much easier to apply, are presented and demonstrated. The functions rely on series manipulation to efficiently generate the coefficients of Taylor series that arise from numerous scenarios involving algebraic, transcendental, and differential/integral operations and compositions with other series.

The major developments and contributions of this research are summarized in the conclusion in Section 6.

2. A Brief Overview of PSM.

In this section, a brief overview of PSM is provided.

Traditionally, using PSM for IVODEs of the form

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(a) = \alpha \tag{2.1}$$

for $a \in \mathbb{R}$ and $\alpha \in \mathbb{R}^{\mathbf{n}}$ required (2.1) to be converted to a polynomial system. Following the development of [10], a polynomial system is an autonomous IVODE of the form

$$\mathbf{x}' = \mathbf{G} \circ \mathbf{x}, \quad \mathbf{x}(a) = \mathbf{b} \tag{2.2}$$

with $a \in \mathbb{R}$, $\mathbf{b} \in \mathbb{R}^{\mathbf{m}}$, $\mathbf{G} : \mathbb{R}^{\mathbf{m}} \longrightarrow \mathbb{R}^{\mathbf{m}}$, and each component of $\mathbf{G}$ a polynomial that is a functional on $\mathbb{R}^m$. Fortunately, the vast majority of IVODEs important to the sciences and engineering may be rewritten in the form of (2.2), (see [10] and [29]). For example, y defined through the nonlinear IVODE

$$y'(t) = \frac{\sin\left(y(t)e^{-t^2}\right)}{\sqrt{t}}, \quad y(1) = 2, \tag{2.3}$$

upon introduction of auxiliary variables,

$$x_1 = y,\ x_2 = \sin\left(ye^{-t^2}\right),\ x_3 = \cos\left(ye^{-t^2}\right),\ x_4 = e^{-t^2},\ x_5 = t,\ x_6 = t^{-\frac{1}{2}}, \tag{2.4}$$

is the first component in the solution of the polynomial system

$$x_1' = x_2 x_6, \qquad x_1(1) = 2 \tag{2.5a}$$

$$x_2' = -2x_1 x_3 x_4 x_5 + x_2 x_3 x_4 x_6, \qquad x_2(1) = \sin\left(2e^{-1}\right) \tag{2.5b}$$

$$x_3' = 2x_1 x_2 x_4 x_5 - {x_2}^2 x_4 x_6, \qquad x_3(1) = \cos\left(2e^{-1}\right) \tag{2.5c}$$

$$x_4' = -2x_4 x_5, \qquad x_4(1) = e^{-1} \tag{2.5d}$$

$$x_5' = 1, \qquad x_5(1) = 1, \tag{2.5e}$$

$$x_6' = -\frac{1}{2}{x_6}^3, \qquad x_6(1) = 1, \tag{2.5f}$$

The polynomial system (2.5) is determined from (2.3) and straightforward differentiation of (2.4). Also, when implementing PSM, these auxiliary variables are often chosen in such a manner to have significance in the study of the model.

Once in polynomial form (2.2), PSM recursively arrives at the power series of $\mathbf{x}$ without the need for explicit differentiation using a simple sequence of Cauchy products, (described subsequently in Section 5). For example, in (2.5), Cauchy products for intermediate variables defined by

$$u_1 = x_2 x_6,\ u_2 = x_3 x_4,\ u_3 = x_1 x_5,\ u_4 = x_2 x_4,\ u_5 = x_4 x_5,\ u_6 = x_6 x_6, \tag{2.6a}$$

$$u_7 = u_6 x_6,\ u_8 = u_2 u_1,\ u_9 = u_2 u_3,\ u_{10} = u_4 u_3,\ u_{11} = u_4 u_1 \tag{2.6b}$$

can be used to calculate the power series of $\mathbf{x}$ recursively. The recursions for the coefficients of the series are initialized by the initial conditions in (2.5) and are given by

$$x_{1\,j+1} = \frac{u_{1\,j}}{j+1}, \quad x_{2\,j+1} = \frac{-2u_{9\,j} + u_{8\,j}}{j+1}, \quad x_{3\,j+1} = \frac{2u_{10\,j} - u_{11\,j}}{j+1}, \tag{2.7a}$$

$$x_{4\,j+1} = \frac{-2u_{5\,j}}{j+1}, \quad x_{5\,j+1} = \begin{cases} 0 & j > 1 \\ 1 & j = 1 \end{cases}, \quad x_{6\,j+1} = -\frac{1}{2}\left(\frac{u_{7\,j}}{j+1}\right), \tag{2.7b}$$

where for notational purposes, it has been assumed that a variable z has a series form of

$$z = \sum_{j=0}^{\infty} z_j (t-c)^j \tag{2.8}$$

with c the center of the series.

With PSM, the coefficients for the 12^{th} degree terms in the series for $\mathbf{x}$ about $t = c$ can be shown to cost less than a mere 150 multiplications/divisions once the recursions have generated the coefficients through the 11^{th} degree terms. Then of course, if desired, once the 12^{th} degree terms are determined, then the 13^{th} degree terms are accessible. In contrast, if one were to use a classic Taylor series method for the IVODE in (2.3) and desired a 12^{th} degree approximation, then to simply compute the the 12^{th} derivative of y needed in the 12^{th} degree term of the Taylor series, repeated differentiation of the right hand side IVODE would be necessary and would result in an expression for $y^{(12)}$ that alone would have over 800

distinct complicated terms, and this derivative (as well as the previous eleven) would need to be evaluated at each step.

Instead of implementing traditional PSM with the polynomial system (2.5), an alternative set of auxiliary variables that use a library of PSM functions developed for this work (see Table A.1 in the appendix; to be explained in Section 5) also may be used to recursively generate the series coefficients for y in (2.3). For example, the auxiliary variables defined by

$$w_1 = -t^2,\ w_2 = e^{-t^2},\ w_3 = ye^{-t^2},\ w_4 = \sin\left(ye^{-t^2}\right), \tag{2.9a}$$

$$w_5 = \cos\left(ye^{-t^2}\right),\ w_6 = t^{-\frac{1}{2}},\ w_7 = \sin\left(ye^{-t^2}\right)t^{-\frac{1}{2}}, \tag{2.9b}$$

have series coefficients, centered at c, recursively generated by the list of functions

$$w_{1\,j} = -\text{powert}\left(\mathbf{w}_{1\,0:j-1}, 2, c\right), \tag{2.10a}$$

$$w_{2\,j} = \text{expy}\left(\mathbf{w}_{2\,0:j-1}, \mathbf{w}_{1\,0:j}\right), \tag{2.10b}$$

$$w_{3\,j} = \text{cauchy\_prod}\left(\mathbf{y}_{0:j}, \mathbf{w}_{2\,0:j}\right), \tag{2.10c}$$

$$w_{4:5\,j} = \text{sincos}\left(\mathbf{w}_{4:5\,0:j-1}, \mathbf{w}_{3\,0:j}\right), \tag{2.10d}$$

$$w_{6\,j} = \text{powert}\left(\mathbf{w}_{6\,0:j-1}, -0.5, c\right), \tag{2.10e}$$

$$w_{7\,j} = \text{cauchy\_prod}\left(\mathbf{w}_{4\,0:j}, \mathbf{w}_{6\,0:j}\right), \tag{2.10f}$$

and this list generates the series coefficients for y through

$$y_{k+1} = \frac{w_{7,\,k}}{k+1}. \tag{2.11}$$

The bold portions in (2.10) represent arrays holding the series coefficients for the respective auxiliary variable. The code list (2.10) has the potential advantage of not requiring to actually form the polynomial systems. Also, the computational cost in using (2.10) versus the more standard PSM approach of (2.5) and (2.6) can be shown to be roughly half the cost. While the functions in Table A.1 were arrived at through PSM techniques, this approach modeled in (2.10) really represents a blend of AD and PSM.

For a desired degree and step size, once either traditional PSM or PSM through our library of functions has recursively calculated the necessary coefficients, the truncated power series is evaluated at the given step, marching the numerical solution forward. These resulting values are then used to initialize the recursions for the next step, much like other traditional explicit one-step methods. However, unlike many traditional methods, including RK methods, PSM generates the coefficients of the Taylor polynomial at each step, so the approximation at each step could be just the evaluation of the polynomial at the step size or the storage of the polynomial itself, allowing for interpolation between step sizes.

3. One-Step Error Analysis and Control.

In this section, the theory that drives an adaptive time-step control is developed. We begin with some preliminaries.

3.1. One-Step Development and Notation.

Suppose that the IVODE,

$$y'(t) = f(t, y(t)), y(a) = \alpha \tag{3.1}$$

is such that f meets the conditions of the Picard-Lindelöf theorem [22] and $f \in C^{n+1}([a,b])$ in t. Then the IVODE (3.1) has a unique solution y that, by Taylor's theorem [8], can be represented as

$$y(t) = \sum_{j=0}^{n} y_j (t-c)^j + \frac{y^{(n+1)}(\xi(t))}{(n+1)!}(t-c)^{n+1} \tag{3.2}$$

for all $c \in (a,b)$ with $\xi(t)$ between c and t. If further, f is real analytic on $[a,b]$, then

$$y(t) = \sum_{j=0}^{\infty} y_j (t-c)^j = y_0 + y_1 (t-c) + y_2 (t-c)^2 + y_3 (t-c)^3 + \cdots \tag{3.3}$$

for all $c \in [a,b]$.

For clarity, we continue for a specific value for n noting that the development extends without loss of generality for any n. For each h such that $t = c + h \in (a,b)$, it follows from (3.2) that, for $n = 5$ e.g.,

$$y(c+h) = y_0 + y_1 h + y_2 h^2 + y_3 h^3 + y_4 h^4 + y_5 h^5 + \frac{y^{(6)}(\xi_1(h))}{6!} h^6 \tag{3.4}$$

where $\xi_1(h)$ is between c and $c+h$. Next, assume for a given c that $u, v \in C^6([a,b])$ are functions that match $y(c+h)$ up to order 4 and order 5, respectively. Such associated functions can be used to obtain results from two distinct numerical methods, or more commonly, methods of different order from the same Runge-Kutta family. (A simple example of a function v for a 2^{nd} order RK method is given below.) By Taylor's theorem and since u matches y through order 4,

$$u(c+h) = u_0 + u_1 h + u_2 h^2 + u_3 h^3 + u_4 h^4 + u_5 h^5 + \frac{u^{(6)}(\xi_2(h))}{6!} h^6 \tag{3.5a}$$

$$= y_0 + y_1 h + y_2 h^2 + y_3 h^3 + y_4 h^4 + u_5 h^5 + \frac{u^{(6)}(\xi_2(h))}{6!} h^6. \tag{3.5b}$$

Similarly since v matches y through order 5, it is useful to record here and is utilized later in Section 3.4, that

$$v(c+h) = v_0 + v_1 h + v_2 h^2 + v_3 h^3 + v_4 h^4 + v_5 h^5 + \frac{v^{(6)}(\xi_3(h))}{6!} h^6 \tag{3.6a}$$

$$= y_0 + y_1 h + y_2 h^2 + y_3 h^3 + y_4 h^4 + y_5 h^5 + \frac{v^{(6)}(\xi_3(h))}{6!} h^6, \tag{3.6b}$$

where $\xi_2(h)$ and $\xi_3(h)$ are again between c and $c+h$. The u_k and v_k in (3.5) and (3.6) represent Taylor coefficients involving derivatives of u and v evaluated at c. We remind the reader that, without loss of generality, the arguments that follow extend to a general degree.

Consider a discretization in time for $t \in [a,b]$ that is possibly non-uniform in step size. Denote this $\{t_0 \equiv a, t_1, \ldots, t_k, \ldots, t_N \equiv b\}$ for integer-valued $N > 0$. We will frequently refer

to the arbitrary subinterval $[t_k, t_{k+1}]$ more explicitly as $[t_k, t_k + h]$ with arbitrary step size h. Given such a time discretization, an arbitrary explicit numerical scheme for an IVODE of the form (3.1) may be denoted

$$w(t_k + h) = w(t_k) + h\varphi(t_k, w(t_k)), \tag{3.7}$$

where φ represents the approximate action of f, and w represents an approximate solution to the exact solution y of (3.1), [28]. Interpolation is then typically used to approximate the solution between these nodes if required.

To provide a simple example of a function u or v as mentioned in (3.5) or (3.6), we examine the classical Heun method, a 2^{nd} order RK method, which for an IVODE (3.1) is given by [28] as

$$y_{k+1} = y_k + \frac{h}{2} f\left(t_k, y_k\right) + \frac{h}{2} f\left(t_{k+1},\ y_k + hf\left(t_k, y_k\right)\right). \tag{3.8}$$

We note that y_k in (3.8) does not represent a Taylor coefficient. Under the theoretical assumption that there is no error in y_k at t_k in (3.8), for the Heun method there is a corresponding function $v(t)$, different for each t_k value, given by

$$v(t) = y(t_k) + \frac{t - t_k}{2} f\left(t_k, y(t_k)\right) + \frac{t - t_k}{2} f\left(t,\ y(t_k) + (t - t_k) f\left(t_k, y(t_k)\right)\right). \tag{3.9}$$

Upon differentiating (3.9) twice and evaluating v, v', and v'' at $t = t_k$ we see that

$$v(t_k) = y(t_k) \tag{3.10a}$$

$$v'(t_k) = f\left(t_k, y(t_k)\right) \tag{3.10b}$$

$$v''(t_k) = \frac{\partial f}{\partial t}\left(t_k, y(t_k)\right) + \frac{\partial f}{\partial y}\left(t_k, y(t_k)\right) f\left(t_k, y(t_k)\right). \tag{3.10c}$$

The values from (3.10) can be used to show that the Taylor polynomials for y, as defined by (3.1) and centered at $t = t_k$, and v from (3.9), also centered at $t = t_k$, match to second degree. So for a given h, there exists $\xi(h)$ such that (compare to the notation of (3.6))

$$v\left(t_k + h\right) = y(t_k) + y'(t_k)h + \frac{y''(t_k)}{2}h^2 + \frac{v'''\left(\xi(h)\right)}{6}h^3 \tag{3.11a}$$

$$= y_0 + y_1 h + y_2 h^2 + \frac{v'''\left(\xi(h)\right)}{6}h^3. \tag{3.11b}$$

Evaluating (3.9) at $t = t_k + h$, we also see that

$$v(t_k + h) = y(t_k) + \frac{h}{2} f\left(t_k, y(t_k)\right) + \frac{h}{2} f\left(t_k + h,\ y(t_k) + hf\left(t_k, y(t_k)\right)\right), \tag{3.12}$$

matching the left-hand side of the Heun method (3.8). Under the assumption that $y(t_k)$ is exact, as will be the case in the local truncation error defined below, $v\left(t_k + h\right)$ in (3.12), and the theoretical equivalent (3.11), is the value provided by Heun's method to approximate

$y(t_k + h)$. Again, following [28], for the Heun method (3.8) and a given h, the function φ in (3.7) would be

$$\varphi(t, y) = \frac{1}{2}\left[f(t, y) + f(t + h, y + hf(t, y))\right]. \tag{3.13}$$

For the RK methods used for comparison purposes in this work (RKF and DP), the expressions similar to (3.9) and (3.13) would be more involved.

3.2. Local Truncation Error. According to classic texts [8, 28], the **local truncation error**, denoted $\tau_{k+1}(h)$, in using an explicit one-step numerical method of the form (3.7) to approximate a solution at time t_{k+1} is

$$\tau_{k+1}(h) \equiv \frac{y(t_k + h) - y(t_k) - h\varphi(t_k, y(t_k))}{h}. \tag{3.14}$$

Definition (3.14) assumes that $y(t_k)$ and $y(t_k + h)$ are exact and contain no error. Also, φ as in (3.7) is specific to the numerical method; for example, φ is given by (3.13) for Heun's method [28]. While (3.14) defines the error locally, the order of $\tau_{k+1}(h)$ matches the global order of the method [8].

If the coefficients $\{u_k\}$ in (3.5) are found by an algorithm that uses only information at t_k and $y(t_k)$, then the expansion in (3.5) becomes

$$u(t_k + h) = y_0 + h\left(y_1 + y_2 h + y_3 h^2 + y_4 h^3 + u_5 h^4 + \frac{u^{(6)}(\xi_2(h))}{6!}h^5\right). \tag{3.15}$$

In the above, $u(t_k + h)$ is computed from some explicit numerical single-step method and is equivalent to the right-hand side of (3.15), where the final two terms involving ξ_2 and u_5 are related to the local truncation error.

We assume at the k^{th} step that $y(t_k)$ would also include error accumulated due to taking multiple steps of the numerical scheme (and roundoff; for this work on our Dell Latitude 3450, machine epsilon is $2.220446049250313 \times 10^{-16}$) and so, in practice, is instead an approximate value, denoted $\bar{y}(t_k)$. Similarly, $\{y_0, y_1, y_2, y_3, y_4\}$ in (3.15) would also be approximate values $\{\bar{y}_0, \bar{y}_1, \bar{y}_2, \bar{y}_3, \bar{y}_4\}$. We note here again that (3.14) assumes $y(t_k)$ and $y(t_k + h)$ are exact and terms involving the bars are not used in the one-step error analysis.

Theoretically, using $u(t_k+h)$, computed from a numerical method, to approximate $y(t_k+h)$ results in a local truncation error from (3.14) of

$$\tau_{k+1}(h) = \frac{y(t_k + h) - y(t_k) - h\left(y_1 + y_2 h + y_3 h^2 + y_4 h^3 + u_5 h^4 + \frac{u^{(6)}(\xi_2(h))}{6!}h^5\right)}{h} \tag{3.16}$$

but $y_0 = y(t_k)$, and so comparing the numerator in (3.16) with (3.4) for $c = t_k$, we see that

$$\tau_{k+1}(h) = (y_5 - u_5)h^4 + \frac{y^{(6)}(\xi_1(h)) - u^{(6)}(\xi_2(h))}{6!}h^5. \tag{3.17}$$

Now if s is a positive scalar, then similar to (3.17), the local truncation error in using $u(t_k + sh)$ via (3.15) to approximate $y(t_k + sh)$ is

$$\tau_{k+1}(sh) = (y_5 - u_5)\, s^4 h^4 + \frac{y^{(6)}(\xi_1(sh)) - u^{(6)}(\xi_2(sh))}{6!} s^5 h^5. \tag{3.18}$$

Since the second term of (3.18) is of higher order in h than the first, the majority of the local truncation error for many IVODEs is typically considered to be associated with the first term, or

$$\tau_{k+1}(sh) = (y_5 - u_5)\, s^4 h^4 + O\left(h^5\right). \tag{3.19}$$

For an estimate of the absolute value of the local truncation error $\tau_{k+1}(sh)$ to be less than some desired error tolerance ε, it would then seem reasonable to require

$$|\tau_{k+1}(sh)| \approx \left|(y_5 - u_5)\, s^4 h^4\right| = \left|(y_5 - u_5)\, h^4\right| s^4 < \varepsilon. \tag{3.20}$$

The inequality in (3.20) is useful and can be manipulated in several ways. In fact, this observation motivates our approach for simple adaptive error estimates in the PSM setting, and is also referenced below in Section 3.4 in our development of the classic scale often used in RK adaptive methods and as presented in [8].

3.3. New PSM Approach to Adaptive Step Size. Recall, the PSM algorithm is designed so that the approximation of $y(t_k + h)$ exactly matches the Taylor polynomial of the solution y to (3.1) expanded about t_k to a given degree n, assuming no error in $y(t_k)$. Viewing PSM as an explicit numerical single-step method of the form represented by (3.7), we have for PSM,

$$w_{k+1} = w_k + h\left(\bar{y}_1 + \bar{y}_2 h + \bar{y}_3 h^2 + \cdots + \bar{y}_n h^{n-1}\right), \tag{3.21}$$

where $\{\bar{y}_1, \bar{y}_2, \bar{y}_3, \cdots, \bar{y}_n\}$ represent the Taylor coefficients of the solution to (3.1) centered at t_k, but based on the approximation w_k to $y(t_k)$, and hence the bar. Note (3.21) is equivalent in theory to the classic higher order Taylor method. The local truncation error defined in (3.14) for (3.21) is

$$\tau_{k+1}(h) \equiv \frac{y(t_k + h) - y(t_k) - h\left(y_1 + y_2 h + y_3 h^2 + \cdots + y_n h^{n-1}\right)}{h}, \tag{3.22}$$

where we remove the bar since we are under the assumption that $y(t_k)$ and $y(t_k + h)$ are exact and contain no error. Proceeding as before with $n = 4$ in (3.21) and using $y_0 = y(t_k)$ and (3.4) with $c = t_k$, we see that (3.22) with (3.4) reduces to

$$\tau_{k+1}(h) = y_5 h^4 + \frac{y^{(6)}(\xi_1(h))}{6!} h^5. \tag{3.23}$$

Again, since the second term of (3.23) is higher order in h than the first, the majority of the local truncation error for many IVODEs would typically be associated with the first term. For $|\tau_{k+1}(h)|$ to be less than some designated error tolerance ε, it would seem reasonable to require

$$|\tau_{k+1}(h)| \approx \left|y_5 h^4\right| < \varepsilon \tag{3.24}$$

or equivalently,

$$|h| < \left|\frac{\varepsilon}{y_5}\right|^{\frac{1}{4}}. \tag{3.25}$$

Some penalty must be paid for ignoring higher order terms [8], so here the step size h is chosen conservatively,

$$h = \pm\left|\frac{\varepsilon}{2y_5}\right|^{\frac{1}{4}} \approx \pm 0.84\left|\frac{\varepsilon}{y_5}\right|^{\frac{1}{4}}, \tag{3.26}$$

where the $\pm$ determines direction.

The value of y_5 in (3.26), in practice, $\bar{y}_5$, is computed with a simple recursion through PSM. We emphasize that there is no such simple calculation of y_5 available for adaptive Runge-Kutta type schemes. RK methods typically approximate y_5 by computing the difference between order 5 and order 6 approximations. (See the related development in Section 3.4). This is prohibitive for general order considerations since changing the order of an RK method requires an entirely new set of complicated coefficients. In contrast, given PSM's easy access to any order, it is a simple matter to generalize (3.26) to apply to an n^{th} order PSMA algorithm. A conservative step analogous to (3.26) for a general n^{th} order PSMA algorithm would thus be given by

$$h = \pm\left|\frac{\varepsilon}{2y_{n+1}}\right|^{\frac{1}{n}}. \tag{3.27}$$

In the case of a system of differential equations, each function would likely produce different h values. The minimum of the h values of all functions in the system at that step is used as the adapted step size. This ensures that the step size is appropriate for all functions in the system to achieve an approximation within the designated error tolerance. Also, if $y_{n+1} = 0$, the step size of h becomes problematic. To avoid such potential issues leading to absurdly large step sizes, a maximum step size may be introduced to maintain accuracy.

3.4. Standard RK Approach to Adaptive Step Size. PSMA was motivated by Fehlberg's approach to a RK adaptive time stepping [14], as presented in [8]. But unlike in the PSM setting, adaptive RK methods depend on a slightly more involved process. Our development is comparable to that presented in [8, 28] as it is driven by Taylor series. Comparing approximate solutions at successive orders, we consider

$$v(t_k + h) - u(t_k + h), \tag{3.28}$$

where u, v are as introduced earlier, with v the higher-order approximation. Using (3.5) and (3.6), with $c = t_k$, the difference (3.28) reduces to

$$v(t_k + h) - u(t_k + h) = (y_5 - u_5)\, h^5 + \frac{v^{(6)}(\xi_3(h)) - u^{(6)}(\xi_2(h))}{6!} h^6. \tag{3.29}$$

Solving (3.29) for $(y_5 - u_5)\, h^4$ yields

$$(y_5 - u_5)\, h^4 = \frac{v(t_k + h) - u(t_k + h)}{h} - \frac{v^{(6)}(\xi_3(h)) - u^{(6)}(\xi_2(h))}{6!} h^5. \tag{3.30}$$

From (3.30) we can see that the middle term in (3.20) is equivalent to

$$\left|(y_5 - u_5)\, h^4\right| s^4 = \left|\frac{v(t_k + h) - u(t_k + h)}{h}\right| s^4 + O\left(h^5\right). \tag{3.31}$$

Then, to $O\left(h^5\right)$,

$$|\tau_{k+1}(sh)| \approx \left|(y_5 - u_5)\, h^4\right| s^4 \approx \left|\frac{v(t_k + h) - u(t_k + h)}{h}\right| s^4, \tag{3.32}$$

which suggests that, for $|\tau_{k+1}(sh)|$ to be roughly less than ε, it would be reasonable to have

$$\left|\frac{v(t_k + h) - u(t_k + h)}{h}\right| s^4 < \varepsilon \tag{3.33}$$

or equivalently,

$$s < \left|\frac{\varepsilon h}{v(t_k + h) - u(t_k + h)}\right|^{\frac{1}{4}}. \tag{3.34}$$

Since some penalty must be paid for ignoring higher order terms, the scale s is again generally chosen conservatively; in fact, for RKF, the typical choice is

$$s = \left|\frac{\varepsilon h}{2\left(v(t_k + h) - u(t_k + h)\right)}\right|^{\frac{1}{4}} \approx 0.84 \left|\frac{\varepsilon h}{v(t_k + h) - u(t_k + h)}\right|^{\frac{1}{4}}, \tag{3.35}$$

(see [8, 28]).

4. Examples. The following examples present comparisons of PSM algorithms with traditionally accepted algorithms that use a RK foundation. We tracked number of steps, accuracy, and Matlab execution time for the standard RK fixed step order 4 method (RK4), DP, and RKF, as well as for PSM fixed step and PSMA. To compare the algorithms without the influence of sophisticated program features intrinsic to Matlab, we coded standalone programs for RK4, RKF, and DP. This provides a controlled setting where all variables, including how a step size is chosen, could be be kept as consistent as possible. Average execution times are presented due to small deviations with runs in Matlab.

The three IVODEs highlighted here are relevant to applied models with real world applications. The first example investigates a straightforward differential equation with a known singularity to compare the various algorithms' abilities to yield an accurate approximation near that singularity. Then we introduce an IVODE that models flight trajectory assuming two degrees of freedom with a focus on the velocity portion of the system. The final example investigates a simple flame propagation model to explore the effect of higher degree approximations on stiffness.

4.1. Example 1: Tangent. Our first example examines a quadratic IVODE in a neighborhood of the domain where the solution becomes singular.

The simple IVODE

$$y' = 1 + y^2, \quad y(0) = 0 \tag{4.1}$$

has the well-known unique explicit solution of

$$y(t) = \tan(t), \tag{4.2}$$

which, of course, has a singularity at $t = \frac{\pi}{2}$. Having access to the exact solution allows for an accurate check of the relative error accrued by a method at a chosen endpoint. For comparison purposes, we tested all algorithms on the designated time interval $[0, 1.57079]$, where the endpoint is close enough to the singularity at $t = \frac{\pi}{2}$ to cause numerical difficulties.

The results follow in Table 4.1. The first methods listed (RK4, RKF, DP) have a standard RK foundation; the fourth-order Runge-Kutta method (RK4) is fixed-step while the others are adaptive. The subsequent methods have a PSM foundation with different specific orders and utilize both fixed and adaptive steps. A designated relative local error tolerance of $\varepsilon = 10^{-11}$ was used with each of the adaptive methods. PSMA was able to converge for $\varepsilon = 10^{-13}$ while $\varepsilon = 10^{-11}$ was the smallest value of ε for which RKF and DP were able to remain numerically stable. The column labeled "Error" indicates the relative error of the approximation to the solution at the last step in the interval. It is noteworthy that none of the standard RK-based algorithms, nor the PSM fixed step algorithms, were able to achieve the desired accuracy and only PSMA succeeded in (and surpassed) the error tolerance of $\varepsilon = 10^{-11}$.

Table 4.1

Singular Example: Comparison of performance in numerical computation of solutions to $y' = 1 + y^2$ on $t \in [0, 1.57079]$, where the true solution becomes singular at $\pi/2$. The performance of standard implementations of RK4, RKF, and DP provide bench marks in step count and execution time. Note for RKF and DP to converge, we had to apply a restriction on the minimum step. These should be compared to the same measures of performance for the PSM and PSMA implementations. PSMA did not require a step size restriction. Adaptive methods used a relative local error tolerance of $\varepsilon = 10^{-11}$.

Method	Error	Steps	Avg. Time (sec)
Runge-Kutta 4	10^{-5}	1000000	2.67
Runge-Kutta-Fehlberg (minstep=10^{-7})	10^{-7}	15992	0.08
Runge-Kutta-Fehlberg (minstep=10^{-9})	10^{-6}	567768	2.83
Dormand-Prince (minstep=10^{-7})	10^{-6}	15377	0.11
PSM Fixed Step *Order 6*	10^{-5}	1000000	1.31
PSM Fixed Step *Order 12*	10^{-9}	1000000	2.62
PSM Fixed Step *Order 24*	10^{-9}	1000000	5.97
PSM Fixed Step *Order 48*	10^{-8}	100000	1.55
PSM Adaptive *Order 6*	10^{-10}	38595	0.08
PSM Adaptive *Order 12*	10^{-11}	593	0.0032
PSM Adaptive *Order 24*	10^{-11}	77	0.001
PSM Adaptive *Order 48*	10^{-12}	28	0.11

The adaptive RK methods also became numerically unstable without the additional requirement of a minimum step size. Without this minimum step size, as the endpoint $t = 1.57079$ was approached, the adaptive RK methods were not able to converge but were trapped in a loop requiring increasingly smaller step sizes. RKF, the most accurate of the two adaptive RK methods tested, took nearly 16000 steps to achieve its approximation with the safety net of a minimum step size of 10^{-7}. The results for PSMA are comparatively impressive and demonstrably better. For example, PSMA Order 24 was able to meet the designated error tolerance in as few as 77 steps and in a fraction of the time of RKF.

While (4.1) is a single example, these results suggest that PSMA should be considered a candidate for solving singular or nearly singular problems. It also strongly suggests that order could make a significant difference with PSMA in accurately and efficiently approximating solutions close to a singularity. We again point out the utility that with PSM schemes, order is controlled through the choice of an upper limit in a loop, so a step-wise change to any order could be easily achieved (though not considered in the work.) Such control is in direct contrast to standard adaptive RK methods where a step-wise change of order requires a new set of increasingly more complicated coefficients, particularly when considering, e.g., a $48^{th} - 49^{th}$ order method.

Timing, while likely to be more stable once implemented in a compiled environment, is anticipated to retain a similar ordering. Notice that the PSMA high order methods are competitive with, and often an order of magnitude faster than, the standard adaptive methods of RKF and DP in addition to maintaining or exceeding the error tolerance capabilities of these standard algorithms, and without the need for a minimum step size.

4.2. Example 2: Two Degrees of Freedom Projectile Equations. Consider the IVODE,

$$y_1' = -\frac{Ac_d}{m}\rho y_1^2 - GM\frac{\sin(y_2)}{y_4^2}, \qquad y_1(0) = 7000 \tag{4.3a}$$

$$y_2' = -GM\frac{\cos(y_2)}{y_1 y_4^2} + \frac{y_1\cos(y_2)}{y_4}, \qquad y_2(0) = \frac{\pi}{4} \tag{4.3b}$$

$$y_3' = \frac{y_1\cos(y_2)}{y_4}, \qquad y_3(0) = \frac{\pi}{4} \tag{4.3c}$$

$$y_4' = y_1\sin(y_2), \qquad y_4(0) = 6.371002\times 10^6, \tag{4.3d}$$

which is a system that models projectile flight within a polar framework. The particle velocity $y_1\,(ms^{-1})$ and the flight path angle y_2 (measured as deviation from the current trajectory) are free variables for the system. The value y_3 is the polar angle measured from initial launch position, and y_4 is the polar radius representing distance in meters from the earth's center.

The environmental parameters in the simulation are the earth's gravitational constant, $G = 6.67408\times 10^{-11}$, the density $\rho = 1$ of air in kilograms per cubic meter, and the mass of the earth in kilograms, $M = 5.972\times 10^{24}$. The projectile parameters are its mass in kilograms, $m = 1000$, the cross sectional area in square meters, $A = 8.75$, and the coefficient of drag, $c_d = 0.5$. For details of this model, see [34].

The system (4.3) cannot be explicitly integrated. The comparative solution for the velocity at $t = 10$ seconds was determined using 40-digit arithmetic with a numerical Taylor Series (TS) package and a 7-8 Runge-Kutta Hall (7-8 RKH) package available in the software Maple. A 28^{th} order Taylor polynomial was required for TS, and for both methods, relative and absolute error tolerances were set to 10^{-27}. The values calculated by these distinct numerical schemes matched to 10^{-29}. Maple's TS package, unlike PSM, calculates the Taylor polynomial exactly symbolically, and 7-8 RKH matches this accuracy by using extremely small step sizes. Each method took approximately a minute in Maple to finish the calculation.

The results in Table 4.2 suggest that PSMA could be an excellent algorithm for projectile simulations and indicate that time step and order appear to matter in both accuracy and timing for these types of applications. The column labeled "Error" indicates the relative error of the approximation to the solution at the last step in the interval. PSMA, again at higher orders, appears recognizably more efficient than the RKF and DP adaptive algorithms, with run time improvements exceeding an order of magnitude, and doing so with significantly fewer steps. We note also in Table 4.2, that adaptive algorithms produce a dramatic drop in the number of steps for both PSM and RK schemes. An interesting additional observation is that PSMA was able to run with the local relative error ε near machine precision. The adaptive algorithms of DP and RKF required local relative error tolerances that could not be pushed smaller than $\varepsilon = 10^{-11}$ and $\varepsilon = 10^{-12}$, respectively, without introducing a minimum step size.

Table 4.2

Projectile Example: Comparison of performance in numerical approximation of velocity $y_1(t = 10)$ in the two degree of freedom particle flight model.The performance of standard implementations of RK4, RKF, and DP provide bench marks in step count and execution time. These should be compared to the same measures of performance for the PSM and PSM adaptive implementations. Adaptive methods used a relative local error tolerance of $\varepsilon = 10^{-11}$.

Method	**Error**	**Steps**	**Avg. Time (sec)**
Runge-Kutta 4	6.68×10^{-15}	100000	1.073
Runge-Kutta-Fehlberg (Adaptive)	1.38×10^{-14}	9331	0.233
Dormand-Prince (Adaptive)	4.22×10^{-13}	8804	0.22
PSM Fixed Step *Order 4*	9.12×10^{-15}	100000	0.69
PSM Fixed Step *Order 8*	9.66×10^{-14}	5000	0.063
PSM Fixed Step *Order 12*	5.26×10^{-15}	5000	0.097
PSM Fixed Step *Order 20*	6.11×10^{-14}	1000	0.037
PSM Fixed Step *Order 60*	8.10×10^{-16}	500	0.092
PSM Adaptive *Order 4*	4.66×10^{-15}	11503	0.139
PSM Adaptive *Order 8*	2.98×10^{-13}	242	0.006
PSM Adaptive *Order 12*	8.71×10^{-15}	85	0.003
PSM Adaptive *Order 32*	3.05×10^{-14}	29	0.014

We mention again that the PSMA produces Taylor splines across its longer steps that are accurate over the entire time step. Therefore, if a value is later desired inside a time step, a simple evaluation using this spline produces a result at the same level of accuracy as

the method. Further, timing continues to appear to be an advantage for PSM techniques. Observe that the PSM fixed step and PSMA higher order methods are often an entire order of magnitude faster than RKF and DP. Again, while it is likely that timing would be more stable when implemented in a compiled environment, it is expected that a similar ordering would be maintained.

4.3. Example 3: The Flame Equation. The third example involves a classic stiff IVODE,

$$y' = y^2 - y^3, \;\; y(0) = \frac{1}{1+e^{\alpha}}, \quad \alpha > 0 \tag{4.4}$$

which is typically analyzed for $t \in [0, 2(1+e^{\alpha})]$. This simple polynomial IVODE models the growth of the radius, $y(t)$, of a match's ball of flame after ignition and the balance of available and consumed oxygen in the application [3]. Although an explicit solution is difficult, an implicit solution of (4.4) is easy to compute, and is given by

$$t = \alpha + e^{\alpha} - \ln\left|1 - \frac{1}{y}\right| + 1 - \frac{1}{y}. \tag{4.5}$$

The solution (4.5) increases quickly near its inflection point $(t^*, 2/3)$ with

$$t^* = \alpha + e^{\alpha} - \ln\frac{1}{2} - \frac{1}{2}, \tag{4.6}$$

and then rapidly asymptotes towards the equilibrium of $y = 1$. This is an example of interest since in a neighborhood around t^*, the solution changes from being non-stiff to stiff. Hence, this is where classic explicit numerical solvers require recognizably more effort to meet a designated error tolerance. It has also been noted that as α increases, so does the computation costs for these methods in this neighborhood. The value of α may be considered a parameter, and the problem becomes increasingly stiff as α increases.

While authors often solve (4.4) across the entire interval $t \in [0, 2(1+e^{\alpha})]$ [3, 24], it is important to note that machine round off error accumulates with each additional step. To minimize this effect, we study (4.4) on a relatively small interval containing the inflection point t^*,

$$[t_0 = \alpha + e^{\alpha} - 3 - e^3, \, t_1 = 2\alpha + e^{\alpha} - e^{-\alpha}], \tag{4.7}$$

instead of over the entire region. The endpoints t_0 and t_1 were determined by (4.5) so that,

$$y(t_0) = \frac{1}{1+e^3}, \;\; y(t_1) = \frac{1}{1+e^{-\alpha}}. \tag{4.8}$$

For a choice of $\alpha = 12$, the interval (4.7) is given by $[t_0 \equiv 162743.7059, t_1 \equiv 162778.7914]$, and was chosen to capture the solution's transition from non-stiff to stiff behavior.

Table 4.3

Flame Example: A comparison of performance in numerical computation of solutions to $y' = y^2 - y^3$ *for* $t \in [t_0 = 162743.7059, t_1 = 162778.7914]$ *with relative error measured at* t_1*. The performance of standard implementations of RK4, RKF, and DP provide benchmarks in step count and execution time. These should be compared to the same measures of performance for the PSM and PSMA implementations. To maintain accuracy with PSMA, we had to restrict the maximum step size to 5. Adaptive methods used a relative local error tolerance of* $\varepsilon = 10^{-13}$*.*

Method	Error	Steps	Avg. Time (sec)
Runge-Kutta-Fehlberg (Adaptive)	2.33×10^{-15}	1630	0.025
Dormand-Prince (Adaptive)	4.52×10^{-14}	1439	0.017
PSM Fixed Step *Order 4*	8.88×10^{-16}	100000	0.144
PSM Fixed Step *Order 8*	2.22×10^{-16}	5000	0.017
PSM Fixed Step *Order 12*	1.11×10^{-16}	1000	0.004
PSM Fixed Step *Order 32*	1.11×10^{-16}	500	0.008
PSM Adaptive *Order 4, maxstep=5*	4.33×10^{-15}	3450	0.034
PSM Adaptive *Order 8, maxstep=5*	6.66×10^{-16}	120	0.001
PSM Adaptive *Order 12, maxstep=5*	1.93×10^{-14}	40	0.0006
PSM Adaptive *Order 20, maxstep=5*	2.55×10^{-15}	17	0.0006
PSM Adaptive *Order 32, maxstep=5*	1.58×10^{-14}	12	0.0005

The results for the adaptive methods for this example compare similarly to the first two examples and are presented in Table 4.3. The column labeled "Error" indicates the relative error of the approximation to the solution at the last step in the interval. A designated relative local error tolerance of $\varepsilon = 10^{-13}$ was used with each of the adaptive methods, and in this example, no minimum step size restriction was additionally required for DP or RKF. The results again suggest the importance of order, specifically in regard to step size (and often run time).

We again believe that the number of steps is the most significant column in Table 4.3, and should be stable across interpreted and compiled environments. We see in Table 4.3 that PSMA at 32^{nd} order can solve (4.4) with only 12 steps, which is a remarkable one hundredth of the steps taken by DP. PSMA also ran over 30 times faster than DP in that scenario. Also, PSMA order 20 requires 17 steps compared to DP's 1439 and RKF's 1630 steps, respectively. The timings for PSM and PSMA higher order methods continue to impress. Again we mention that while we expect timings to be more stable when implemented in a compiled environment, it is likely that timings would retain a similar ordering. This example indicates that the higher order approximations available with explicit PSM techniques may have a positive impact on stiff or nearly stiff IVODEs.

5. Library of PSM Functions. A drawback to PSM techniques has been the need for the IVODEs to be manually written as a polynomial system. This section details the development of a growing set of PSM functions that perform PSM recursions corresponding to each transcendental function or nonlinear operation that arises in a large class of IVODEs. See Table A.1 in the appendix for a list of many of the PSM functions derived. The PSM recursions

performed by these functions are equivalent to those given by some systems of polynomial differential equations, however they do not require actually forming the polynomial systems. Instead, they require manual decomposition of the IVODEs into a code list form. We see this as an efficient hybrid between pure PSM (manual conversion of IVODEs to polynomial form) and pure AD (computational conversion to series recursion formulas) by coding auxiliary variables in a list of steps with one function or (nonlinear) operation per step and calling a library of corresponding recursion formulas.

Fortunately, the vast majority of differential equations relevant to the engineering and scientific communities, even those that are highly nonlinear, can be recast in polynomial form [10]. It is typically straightforward to convert a differential equation into polynomial form. It is equivalently straightforward to manually decompose an IVODE into a code list form where operations are handled through calls to our PSM library of functions. A simple example contrasting regular PSM through polynomial systems and PSM through our library of functions was given in Section 2. Both techniques have the advantage that they recursively generate the coefficients of the power series to a desired degree without taking successive complicated derivatives.

Computationally, both traditional PSM and PSM through our library of functions, after initialization, rely solely on basic arithmetic operations and do not use MATLAB's transcendental functions. Compared to other methods, this has advantages in timing and efficiency for many IVODEs. Also, the functions in this library encompass standard parent functions as well as more complex compositions of functions. In applying our PSM technique to various IVODEs, the prescribed code list calls upon functions from this library to generate the Taylor polynomial for the approximation, which can then be evaluated at a designated time step to yield an approximation to the solution.

We briefly survey some of the basic theory behind PSM, and demonstrate an example illustrating the method used to generate the recursive relationships that form the foundation of functions we have implemented.

5.1. Basic PSM Framework and Construction of Subfunctions. In the late 1980s, Parker and Sochacki recognized that if IVODEs were in polynomial form, the first n terms of the n^{th} Picard iterate was the Taylor polynomial [29]. Thus, they further recognized that Picard iteration could be used as a Taylor generator that bypassed the traditional need for explicit differentiation. Computational enhancements to PSM came when it was further shown that the recursive Picard iterations could be computed without the need of integration. The framework for this approach is motivated by power series. If the series of an auxiliary variable y centered at a step t_k is given by

$$y(t) = \sum_{j=0}^{\infty} y_j (t - t_k)^j, \tag{5.1}$$

then y' has the expansion

$$y'(t) = \sum_{j=0}^{\infty} (j+1) y_{j+1} (t - t_k)^j. \tag{5.2}$$

If the first n terms of the series for the polynomial system (as an example, the system (2.5) corresponding to the IVODE (2.3)) are known, then the $(n+1)^{st}$ term can be determined from the right-hand side of the system (see e.g., (2.7)). This computational PSM iterative scheme is a robust extension of the classic power series method.

Since the systems are polynomial, there is a need to multiply series with this technique, and so the Cauchy product plays a significant role in the method. Recall that if the coefficients of the series of a and b are known up to degree n, then the n^{th} degree coefficient of the series for the product $c = ab$ can be found by

$$c_n = \sum_{j=0}^{n} a_j b_{n-j}, \tag{5.3}$$

a simple dot product with the coefficient array of a and the transpose of b.

The Cauchy product (5.3) is a key component in the library of functions. The user simply inputs the desired degree n for the Taylor coefficient of the product c and two arrays containing the coefficients of a and b at least up to n, and the function uses a basic loop to calculate the desired coefficient c_n. See A.4 in the appendix for the psuedocode of **cauchy_prod.m**. Computational costs for PSM techniques are often measured in number of Cauchy products required in an algorithm.

5.2. Example of a Development of a PSM Recursion in the Library. As an example of the method used to implement many of the algorithms in the library of functions, we develop the recursion used for the **expy** function; the calculation of the n^{th}degree Taylor coefficient of the natural exponential of a power series. Given the coefficients of the power series for y to at least degree n, **expy** can be used to recursively generate the coefficients of the power series to degree n of

$$w = e^y. \tag{5.4}$$

We derive **expy**, by assuming y is expanded as in (5.1) and w similarly as

$$w(t) = \sum_{j=0}^{\infty} w_j (t - t_j)^j. \tag{5.5}$$

Differentiating (5.4) and then substituting,

$$w' = e^y\, y' \tag{5.6a}$$

$$= y' w. \tag{5.6b}$$

Note that an IVODE for y provides the necessary recursion to compute enough terms of y to determine the next term in w. The form in the second equation in (5.6) suggests the use of the Cauchy product (5.3), and assuming an expansion (5.2) for y', and an analogous expansion for w', matching n^{th} degree coefficients leads to

$$(n+1) w_{n+1} = \sum_{j=0}^{n} (j+1)\, y_{j+1} w_{n-j}. \tag{5.7}$$

With a slight shift of an index on w, the n^{th} degree Taylor coefficient of w is calculated by

$$w_n = \frac{1}{n}\left(\sum_{j=0}^{n-1}(j+1)\,y_{j+1}w_{n-1-j}\right). \tag{5.8}$$

Equation (5.8) can be coded as **expy**. The user merely inputs n, an array with the series coefficients of w to degree $n-1$, and an array with the known coefficients of y to at least degree n, and with a Cauchy product the function efficiently outputs the n^{th} degree coefficient of w.

Most functions currently included in the library follow a similar development, and as in (5.8), the function returns a coefficient of the Taylor polynomial to a desired degree whose calculation depends strictly on previously determined lower degree coefficients. Working together, these functions can be used to efficiently generate extremely high order approximations to many dynamical models. See Table A.1 for a list of functions that were implemented in Matlab, as well as syntax required to call them.

5.3. Examples Using Functions Implemented. Pseudocode for algorithms (A.1, A.2, and A.3) is provided in the Appendix for the three IVODEs considered in Section 4, equations (4.1), (4.3), and (4.4), respectively. These nonlinear problems are easily coded using functions drawn from the library table A.1, and these calls are highlighted in blue. A pseudo-code implementation of the adaptive time stepping control (A.5) is also included.

6. Conclusion. This work focused on the development and testing of a novel adaptive PSM algorithm which we call PSMA for PSM with adaptive stepping. This method is similar to a Taylor method, but it uses a simple computational approach to recursively generate the coefficients of a power series to a desired degree without taking extensive high order derivatives. In Section 3, the theory for regular adaptive stepping and local truncation error is established. The theory used to develop PSMA is adapted from the recognized adaptive methods with a RK foundation. Adaptive stepping was incorporated into the PSM algorithms, and the classic adaptive scheme for the standard RK algorithms was treated in brief. In comparing these PSM algorithms to the classic methods, PSM and PSMA have realized several useful theoretical, computational, and practical gains in generating highly accurate solutions in the IVODE setting. First, a beneficial feature of PSMA is its ability to recursively generate an approximation to the y_{k+1} coefficient given approximations to the previous coefficients, which is not true for most standard adaptive numerical solvers. Also, order in PSMA is increased by changing a single, order-control parameter. Since PSM can recursively generate Taylor coefficients of arbitrary order, the estimate on which the PSMA time step is based is therefore easy to both construct and implement, and is easy to apply to arbitrary order. This would be far harder for solvers of the RK family as an increase of order for the RK family, for instance, instead requires a new set of weights as indicated by the Butcher Tableau for the method.

In Section 4, the PSM algorithms (fixed step and adaptive stepping) and the standard RK foundation algorithms (RK4, RKF, DP) are applied to three illustrative examples in order to compare the accuracy, number of steps, and timings in approximating the solutions. All three nonlinear examples suggest that PSMA is both effective and efficient. The first

example, containing a singular solution, examined the potential effectiveness of higher order approximations available with PSMA for studying singular problems. The second example was an IVODE used in simulations modeling flight trajectories of a projectile [34] to explore the accuracy and efficiency of the algorithms in an applied setting. The third example examined the effect of higher order approximations and investigated the behavior of the algorithms in a flame propagation model with a stiff solution. In each example, PSMA was faster, achieving similar or better accuracy in considerably fewer time steps in comparison to fixed-stepped PSM methods of equal order, as expected. PSMA also proved to be more efficient when compared to traditional explicit fixed step methods RK4 and the adaptive methods RKF and DP, particularly at higher orders. Examples hint that a computational minimum in (time, order) space may exist, and that the order of approximation is very important for some regions.

Lastly, in Section 5, the basic structure of PSM's iterative scheme was explained, including the PSM specific functions that perform PSM recursions for operations that appear in a large class of IVODEs. The functions require decomposing the IVODEs into a code list form, but bypass the need to convert IVODEs into a polynomial system, which is a requirement and potential obstacle for regular PSM. These functions allow the coefficients of a power series to be recursively generated to an arbitrary degree. Also included in this section are a sample derivation of the recursion and pseudo-code implementations for the examples of Section 4. All of the functions created during this project are listed in Table A.1. The functions in this library were used in pseudocodes A.1 ,A.2, and A.3 for the examples from Section 4. The goal is to grow this library of functions so that it might serve as part of the foundation for a future large-scale PSM tool for the scientific and engineering communities.

The future of PSMA includes further increasing its computational efficiency by moving from its present Matlab exploratory environment to a compiled platform. Beyond this, and as noted in [31], the Cauchy product framework of PSM makes it well-suited for parallelization. As such, potential gains in parallelizing PSMA will also be an important direction. In addition, PSMA is currently adaptive only in step size, but efficiency could be gained if we could also adapt in order. This is another potential benefit of our methods as compared to RK-based methods, since "on-the-fly" movement between orders for the latter cannot be easily accommodated. Lastly, since PSM generates the Taylor polynomial at each step, and the Taylor polynomial is a starter for other numerical methods, it is worth exploring the possibility of other PSM adaptive methods. In particular, while fixed step Padé approximants were briefly considered in [10], adaptive Padé approximants with PSM could potentially produce similar accuracy to PSMA with a significant reduction in the number of steps, and possibly in less time. This potential highlights yet another research direction that is poised to build from this work.

Acknowledgments. Work for this research was funded in part through the James Madison University Jeffrey E. Tickle '90 Family Endowment housed in the College of Science and Mathematics. We would also like to thank all of our advisors from the PSM group at JMU! Lastly, we extend our gratitude to the referees of this work who provided us with detailed comments in the reviews. We greatly appreciate their deep insight into helping us make the final version of this manuscript notably clearer and more effective.

REFERENCES

[1] I. M. Abdelrazik and H. A. Elkaranshawy, *Extended Parker Sochacki method for Michaelis-Menten enzymatic reaction model*, Analytic Biochemistry, 496 (2016), pp. 50–54, https://doi.org/10.1016/j.ab.2015.11.017.

[2] I. M. Abdelrazik, H. A. Elkaranshawy, and A. M. Abdelrazek, *Modified Parker-Sochacki method for solving nonlinear oscillators*, Mechanics Based Design of Structures and Machines, 45 (2017), pp. 239–252, https://doi.org/10.1080/15397734.2016.1201425.

[3] S. Abelman and K. C. Patidar, *Comparison of some recent numerical methods for initial-value problems for stiff ordinary differential equations*, Comput. Math. Appl., 55 (2008), pp. 733–744, https://doi.org/10.1016/j.camwa.2007.05.012.

[4] S. O. Akindeinde, *Improved Parker-Sochacki method for closed form solution of two and three-point boundary value problems of nth order odes*, Int. J. Appl. Math., 29 (2016), pp. 597–607, https://doi.org/10.12732/ijam.v29i5.7.

[5] S. O. Akindeinde, *Parker-Sochacki method for the solution of convective straight fins problem with temperature-dependent thermal conductivity*, Int. J. Nonlinear Sci., 25 (2018), pp. 119–128.

[6] S. O. Akindeinde and E. Okyere, *New analytic technique for the solution of nth order nonlinear two-point boundary value problems*, British Journal of Mathematics and Computer Science, 15 (2016), pp. 1–11, https://doi.org/10.9734/BJMCS/2016/24365.

[7] D. Barton, I. Willers, and R. Zahar, *Automatic solution of systems of ordinary differential equations by the method of taylor series*, Comput. J., 14 (1971), pp. 243–248.

[8] R. L. Burden and J. D. Faires, *Numerical Analysis: 4th Ed*, PWS Publishing Co., Boston, MA, 1989.

[9] D. C. Carothers, S. K. Lucas, G. E. Parker, J. D. Rudmin, J. S. Sochacki, R. J. Thelwell, A. Tongen, and P. G. Warne, *Connections between power series methods and automatic differentiation*, Recent Advancements in Algorithmic Differentiation, 87 (2012), pp. 175–186, https://doi.org/10.1007/978-3-642-30023-3_16.

[10] D. C. Carothers, G. E. Parker, J. S. Sochacki, and P. G. Warne, *Some properties of solutions to polynomial systems of differential equations*, Electron. J. Differential Equations, 2005 (2005), pp. 1–17.

[11] L. Chen and D. Junsheng, *Multistage numerical Picard iteration methods for nonlinear Volterra integral equations of the second kind*, Advances in Pure Mathematics, 5 (2015), p. 672.

[12] P. S. Dwyer, *Interval analysis: by ramon e. moore. 145 pages, diagrams, 6 9 in. new jersey, englewood cliffs, prentice-hall, 1966. price, 9.00*, J. Franklin inst., 284 (1967), pp. 148–149.

[13] H. A. Elkaranshawy, A. M. Abdelrazek, and H. M. Ezzat, *Power series solution to sliding velocity in three-dimensional multibody systems with impact and friction*, International Journal of Mathematical, Computational, Physical, Electric, and Computer Engineering, 9 (2015), pp. 585–588.

[14] E. Fehlberg, *Numerical integration of differential equations by power series expansions, illustrated by physical examples*, Technical Report NASA-TN-D-2356, NASA, 1964.

[15] C. W. Gear, R. D. Skeel, and S. G. Nash, *The development of ODE methods: A symbiosis between hardware and numerical analysis*, in A History of Scientific Computing, ACM Press, 1990.

[16] A. M. Gofen, *The ordinary differential equations and automatic differentiation unified*, Complex Var. Elliptic Equ., 54 (2009), pp. 825–854, https://doi.org/10.1080/17476930902998852.

[17] M. Hopkins and S. Furber, *Accuracy and efficiency in fixed-point neural ODE solvers*, Neural Comput., 27 (2015), pp. 2148–2182, https://doi.org/10.1162/NECO_a_00772.

[18] A. Jorba and M. Zou, *A software package for the numerical integration of odes by means of high-order taylor methods*, Exp. Math., 14 (2005), pp. 99–117.

[19] T. Kimura, *On dormand-prince method*. Japan Malaysia Technical Institute, 2009.

[20] J. Liu, G. E. Parker, J. S. Sochacki, and A. Knutsen, *Approximation methods for integrodifferential equations*, Proceedings of the International Conference on Dynamical Systems and Applications, III (2001), pp. 383–390.

[21] J. Liu, J. S. Sochacki, and P. Dostert, *Singular perturbations and approximations for integrodifferential equations*, in Differential Equations and Control Theory, S. Aizicovici and N. H. Pavel, eds., CRC Press, 2001.

[22] I. M. Mack, *Generalized Picard-Lindelöf Theory*, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 1991.

[23] N. Nakhjiri and B. Villac, *Modified Picard integrator for spaceflight mechanics*, Journal of Guidance, Control, and Dynamics, 37 (2014), pp. 1625–1637, https://doi.org/10.2514/1.G000303.
[24] M. Nechita, *Revisiting a flame problem. remarks on some non-standard finite difference schemes*, Didactica Mathematica, 34 (2016), pp. 51–56.
[25] R. D. Neidinger, *Introduction to automatic differentiation and matlab object-oriented programming*, SIAM Rev., 52 (2010), pp. 545–563, https://doi.org/10.1137/080743627.
[26] E. Nurminskii and A. Buryi, *Parker-Sochacki method for solving systems of ordinary differential equations using graphics processors*, Numer. Anal. Appl., 4 (2011), p. 223, https://doi.org/10.1134/S1995423911030049.
[27] B. S. Ogundare, S. O. Akindeinde, A. O. Adewumi, and A. A. Aderogba, *Improved Parker-Sochacki approach for closed form solution of enzyme catalyzed reaction model*, J. Mod. Methods Numer. Math., 8 (2017), pp. 90–98, https://doi.org/10.20454/jmmnm.2017.1251.
[28] J. M. Ortega, *Numerical analysis: A second course*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.
[29] G. E. Parker and J. S. Sochacki, *Implementing the Picard iteration*, Neural Parallel Sci. Comput., 4 (1996), pp. 97–112.
[30] C. D. Pruett, W. H. Ingham, and R. D. Herman, *Parallel implementation of an adaptive and parameter-free n-body integrator*, Comput. Phys. Commun., 182 (2011), pp. 1187–1198, https://doi.org/10.1016/j.cpc.2011.01.014.
[31] C. D. Pruett, J. W. Rudmin, and J. M. Lacy, *An adaptive N-body algorithm of optimal order*, J. Comput. Phys., 187 (2003), pp. 298–317, https://doi.org/10.1016/S0021-9991(03)00101-3.
[32] J. W. Rudmin, *Application of the Parker-Sochacki method to celestial mechanics*, technical report, James Madison University, 1998.
[33] A. Singarimbun, Y. Fujimitsu, M. Djamal, and R. Dewi, *Pressure transient modeling in geothermal reservoir by using Picard-Mclaurin iteration*, Advanced Materials Research, 1025–1026 (2014), pp. 959–973, https://doi.org/10.4028/www.scientific.net/AMR.1025-1026.959.
[34] G. M. Siouris, *Missile guidance and control systems*, Springer, New York, NY, 2011.
[35] E. Smirnov and E. Timoshkova, *Comparative investigation of methods for the numerical prediction of motion of asteroids that approach the Earth: Example of the 99942 Apophis asteroid*, Cosmic Research, 52 (2014), pp. 118–124, https://doi.org/10.1134/S0010952514020075.
[36] J. S. Sochacki, *Polynomial ODEs: Examples, solutions, properties*, Neural Parallel Sci. Comput., 18 (2010), pp. 441–450.
[37] R. D. Stewart and W. Bair, *Spiking neural network simulation: numerical integration with the Parker-Sochacki method*, J. Comput. Neurosci., 27 (2009), pp. 115–133, https://doi.org/10.1007/s10827-008-0131-5.
[38] R. J. Stewart and K. N. Gurney, *Spiking neural network simulation: memory-optimal synaptic event scheduling*, J. Comput. Neurosci., 30 (2011), pp. 721–728.
[39] P. Szynkiewicz, *A novel GPU-enabled simulator for large scale spiking neural networks*, Journal of Telecommunications and Information Technology, 2 (2016), pp. 34–42.
[40] P. G. Warne, D. A. Warne, J. S. Sochacki, G. E. Parker, and D. C. Carothers, *Explicit a-priori error bounds and adaptive error control for approximation of nonlinear initial value differential systems*, Comput. Math. Appl., 52 (2006), pp. 1695–1710, https://doi.org/10.1016/j.camwa.2005.12.004.

Appendix A. Library of PSM Functions and Pseudocode.

This appendix includes Table A.1 containing a list of functions that were implemented in Matlab, as well as pseudocode syntax for calling them. We utilize an array index zero for coefficients rather than the Matlab index in order to agree with the degree of the term. Pseudocode for the algorithms denoted below as A.1 ,A.2, and A.3 for the three IVODEs considered in Section 4 (4.1, 4.3, and 4.4, respectively) is also outlined.

$f(t) = \sum_{k=0}^{\infty} f_k (t - c_0)^k$	$f(y(t)) = \sum_{k=0}^{\infty} [f(y)]_k (t - c_0)^k$
$[f(t) = t^p]_k \leftarrow$ **powert**(fcoeff(0:k-1),p,c0)	$[f(y) = y^p]_k \leftarrow$ **powery**(fcoeff(0:k-1),ycoeff(0:k),p)
$[f(t) = e^t]_k \leftarrow$ **expt**(fcoeff(0:k-1),c0)	$[f(y) = e^y]_k \leftarrow$ **expy**(fcoeff(0:k-1),ycoeff(0:k))
$[f(t) = C]_k \leftarrow$ **const**(C,k)	$[f(y) = r^y]_k \leftarrow$ **rpowy**(fcoeff(0:k-1),ycoeff(0:k))
$[f(t) = \ln(t)]_k \leftarrow$ **natlogt**(fcoeff(0:k-1))	$[f(y) = \ln(y)]_k \leftarrow$ **natlogy**(fcoeff(0:k-1),ycoeff(0:k))
$[f(t) = \cos(t)]_k \leftarrow$ **cost**(fcoeff(0:k-1),c0)	
$[f(t) = \sin(t)]_k \leftarrow$ **sint**(fcoeff(0:k-1),c0)	$\left[f(y) = \frac{\sin(y)}{\cos(y)}\right]_k \leftarrow$ **sincos**(fcoeffs(0:1,0:k-1)
$[f(t) = \tan(t)]_k \leftarrow$ **tant**(fcoeff(0:k-1),c0)	$[f(y) = \tan(y)]_k \leftarrow$ **tany**(fcoeff(0:k-1),ycoeff(0:k))
$[f(t) = \cot(t)]_k \leftarrow$ **cott**(fcoeff(0:k-1),c0)	$[f(y) = \cot(y)]_k \leftarrow$ **coty**(fcoeff(0:k-1),ycoeff(0:k))
$[f(t) = \sec(t)]_k \leftarrow$ **sect**(fcoeff(0:k-1),c0)	$[f(y) = \sec(y)]_k \leftarrow$ **secy**(fcoeff(0:k-1),ycoeff(0:k))
$[f(t) = \csc(t)]_k \leftarrow$ **csct**(fcoeff(0:k-1),c0)	$[f(y) = \csc(y)]_k \leftarrow$ **cscy**(fcoeff(0:k-1),ycoeff(0:k))
f(t) = Horners(f,t) $\leftarrow$ **Horners**(fcoeff(0:k),t)	$[f = a \cdot b]_k \leftarrow$ **cauchy_prod**(acoeff(0:k),bcoeff(0:k))
	$[f = \frac{a}{b}]_k \leftarrow$ **div**(acoeff(0:k),bcoeff(0:k),fcoeff(0:k-1))

Table A.1
Library of Functions

Algorithm A.1 Example 3.1: Tangent

```
1:  procedure FIND TAN([t0,tend],x0,tol,deg,minstep)        ▷ solve x' = 1 + x^2
2:      [xcoeff] = Initialize(deg)                          ▷ initialize 1×deg xcoeff matrix
3:      xcoeff(0) = x0                                      ▷ apply ICs
4:      while tcurrent < tend do                            ▷ run for full interval
5:          for k ← 0, deg do                               ▷ step through degree
6:              fkth = const(1,k)+cauchy_prod(xcoeff(0:k),xcoeff(0:k))   ▷ [1 + x^2]_k
7:              xcoeff(k+1) = fkth/(k+1)                    ▷ integrate [f]_k wrt t
8:          end for
9:          tstep = psm_step(tol,xcoeff,minstep)            ▷ use psm bound
10:         tstep = min(tstep,tend)                         ▷ check and fix for end of interval
11:         xfinal = Horner(xcoeff,tstep)                   ▷ evaluate at end of subinterval
12:         xcoeff(0) = xfinal                              ▷ initialize x0 for next time iteration
13:         tcurrent = tcurrent + tstep                     ▷ update current time value
14:     end while
15:     return xfinal
16: end procedure
```

Algorithm A.2 Example 3.2: 2DOF Projectile Motion

```
1: procedure MissileODE([t0,tend],y0,tol,deg,parameters, step_opts)        ▷ solve (4.3)
2:     [ycoeff,ucoeff] = Initialize(deg)  ▷ initialize 4×deg ycoeff and 10×deg auxillary ucoeff matrix
3:     ycoeff(1:4,:) = y0(1:4,parameters)                                  ▷ apply ICs
4:     while tcurrent < tend do                                            ▷ run for full interval
5:         for k ← 1, deg do                                               ▷ step through degree
6:             ucoeff(1,k) = div(ycoeff(1,0:k),ycoeff(4,0:k),ucoeff(0:k-1))          ▷ y1/y4
7:             ucoeff(2:3,k) = sincosy(ucoeff(2:3,0:k-1),ycoeff(2,0:k))  ▷ sin(y2) , cos(y2) eval
8:             ucoeff(4,k) = cauchy_prod(ycoeff(4,0:k),ycoeff(4,0:k))                ▷ y4^2
9:             ucoeff(5,k) = cauchy_prod(ycoeff(1,0:k),ucoeff(4,0:k))                ▷ y1 · y4^2
10:            ucoeff(6,k) = cauchy_prod(ucoeff(1,0:k),ucoeff(3,0:k))        ▷ y1 · cos(y2)/y4
11:            ucoeff(7,k) = cauchy_prod(ycoeff(1,0:k),ucoeff(2,0:k))        ▷ y1 · sin(y2)
12:            ucoeff(8,k) = div(ucoeff(2,0:k),ucoeff(4,0:k),ucoeff(8,0:k-1))        ▷ sin(y2)/y4^2
13:            ucoeff(9,k) = div(ucoeff(3,0:k),ucoeff(5,0:k),ucoeff(9,0:k-1))   ▷ cos(y2)/(y1 y4^2)
14:            ucoeff(10,k) = cauchy_prod(ycoeff(1,0:k),ycoeff(1,0:k))               ▷ y1^2
15:            % Assemble RHS using parameters
16:            f(1) = -A*cd*rho/m*ucoeff(10,k) - G*M*ucoeff(8,k)                     ▷ (4.3a)
17:            f(2) = -G*M*ucoeff(9,k) + ucoeff(6,k)                                 ▷ (4.3b)
18:            f(3) = ucoeff(6,k)                                                    ▷ (4.3c)
19:            f(4) = ucoeff(7,k)                                                    ▷ (4.3d)
20:            ycoeff(1:4,k+1) = f(1:4)/(k+1)                  ▷ integrate [fvec]_k wrt t
21:        end for
22:        [tstep] ← psm_step                                  ▷ use psm bound and adjust step
23:        yfinal_vec = Horner(ycoeff(:,0:k),tstep)            ▷ Evaluate coefficients
24:        ycoeff(:,0) = yfinal_vec                            ▷ Update for next time interval loop
25:        tcurrent = tcurrent + tstep                         ▷ update current time value
26:    end while
27:    return yfinal_vec
28: end procedure
```

Algorithm A.3 Example 3.3: Flame equation

```
1: procedure FLAMEODE([t0,tend],x0,tol,deg,minstep)        ▷ solve x' = x^2 − x^3
2:     [xcoeff] = Initialize(deg)                          ▷ initialize 1×deg xcoeff matrix
3:     xcoeff(0) = x0                                      ▷ apply ICs
4:     while tcurrent < tend do                            ▷ run for full interval
5:         for k ← 0, deg do                               ▷ step through degree
6:             x2k = cauchy_prod(xcoeff(0:k),xcoeff(0:k))  ▷ [y^2]_k
7:             x3k = cauchy_prod(xcoeff(0:k),x2(0:k))      ▷ [y^3]_k
8:             fkth = x2k - x3k;                           ▷ [y^2 − y^3]_k
9:             xcoeff(k+1) = fkth/(k+1)                    ▷ integrate [f]_k wrt t
10:        end for
11:        tstep = psm_step(tol,xcoeff,minstep)            ▷ use psm bound
12:        tstep = min(tstep,tend)                         ▷ check and fix for end of interval
13:        xfinal = Horner(xcoeff,tstep)                   ▷ evaluate at end of subinterval
14:        xcoeff(0) = xfinal                              ▷ initialize x0 for next time iteration
15:        tcurrent = tcurrent + tstep                     ▷ update current time value
16:    end while
17:    return xfinal
18: end procedure
```

Algorithm A.4 Cauchy Product

```
1: procedure CAUCHY_PROD(acoeff,bcoeff)                    ▷ implement (5.3)
2:     deg=length(acoeff)
3:     out = dot(acoeff(0:deg), flip(bcoeff(0:deg)))       ▷ acoeff ≡ bcoeff is special case
4:     return out
5: end procedure
```

Algorithm A.5 PSM Adaptive Step Size

```
1: procedure PSM_STEP(tol,coeffs,optional_vars)            ▷ Use our estimate for time step
2:     [varcount,deg+1] ← size(coeffs)
3:     for k ←1,varcount do                                ▷ Step through all variables
4:         hvec(k) ← abs(tol/2/coeffs(k,deg+1))**(1/deg)   ▷ apply (3.27)
5:     end for
6:     h ← min(hvec)                                       ▷ min step over all variables
7:     h ← check h and fix for minstep & maxstep options   ▷ use optional_vars
8:     return h
9: end procedure
```