AUTOMATIC SERIES RECURRENCE RELATIONS FOR ORDINARY DIFFERENTIAL EQUATIONS

RICHARD D. NEIDINGER
DAVIDSON COLLEGE

Abstract. An automatically generated recurrence algorithm can compute the coefficients of the power series solution of any ordinary differential equation(s) (ODEs) as given in MATLAB for numerical solution. The algorithm has a variable initial point to any variable order, allowing large step sizes in a numerical high-order Taylor series method. This algorithmic approach has been known for decades but is introduced and implemented here for a broad audience. A program writes the MATLAB code for the recurrence algorithm, clearly readable evidence that power series solutions are possible for any (including nonlinear) ODEs. This gives a high-order Taylor polynomial solution that is good within most of the radius of convergence. Tests show that stepping produces piecewise polynomial solutions that can achieve high accuracy with far fewer steps than usual methods, ode45 in particular. The program that writes the recurrence uses operator overloading in object-oriented programming in MATLAB, as is explained.

1. Introduction. This article presents an accessible introduction to a modern algorithmic approach to power series solutions for differential equations and includes implementation details for the numerical method. We focus on forming, viewing, and then using the recurrence algorithm for series coefficients in both pseudo code and in automatically generated MATLAB code. The generating program writes a series recurrence relation, as an algorithm, for any ordinary differential equation (ODE) or system of ODEs about a variable initial condition, assuming it can be written as a system of first order ODEs (as is typically done for numerical solutions) with coefficient functions that are analytic in some nontrivial interval about the initial condition. We seek to broaden awareness and understanding of this algorithmic Taylor method that has been around for decades in different versions and that researchers have found to be useful (e.g. [2]). The method contrasts with many traditional treatments of power series solutions in differential equations (e.g. [3]) and of higher-order Taylor methods in numerical analysis (e.g. [4]) that still give the impression that symbolic manipulations are required, either by hand or using symbolic computation. We use the terms power series and Taylor series interchangeably with the understanding that expansion about an arbitrary point is necessary for steps in an ODE solver.

The algorithmic Taylor method dates back to at least 1960 [8] and is well-known in the automatic differentiation research community [18] [13] [10]. Specialized software tools, for solving ODEs using this method, date from the last century [6] [5] to more recent times [12] [1]. Implementations can be complex, relying on either source transformation (reading and writing text files for an ODE and series respectively) or object-oriented programming (OOP) that overloads the operations in the ODE. Another research community [17] [23] [20] has taken a related approach to bridging the gap between an ODE and the recurrence relation algorithm for the power series solution. They focus on rewriting any ODE system as a system of only polynomial (or just quadratic) expressions for each derivative, as we mention in Section 2.

Our software package (`https://github.com/rineidinger/psm4odes`) uses OOP to accomplish source code transformation in MATLAB, a platform that is already used in many numerical analysis classes and other applied settings. Users can specify the ODE by the same file that would be used for built-in solvers such as `ode45`. The sections in this paper can be used to appreciate the algorithmic Taylor method on several different levels. Section 2 is an elementary introduction, that could be used as

an alternative way to present power series solutions in differential equations. Section 3 provides a summary of the foundational recurrences behind the method [13] [22] [16] [10]; this section can be skipped or skimmed and returned to as a reference as needed. Section 4 focuses on the output MATLAB code that is the series recurrence algorithm for a given ODE at variable initial condition to variable order. This tangible result could help develop appreciation for a computational approach to power series solutions or the high-order Taylor method. Practically, executing this code is more efficient (and would be even more so in a compiled language) than an alternative that directly overloads the operators to perform (instead of write) the recurrence step(s). Seeing the recurrence code can help bridge the gap in many black-box solvers from specifying an ODE to the computed solution. Section 5 shows how object-oriented programming with operator overloading is used to turn operations done in the ODE into character strings for each recurrence that are assembled into the output series recurrence file. This section can be skipped for readers interested in using, not implementing, the method. However, such details may be a highlight for those interested in computing details that are often omitted with other packages. The ideas could be used in other languages, perhaps Python for an alternative exposition or C++ for computational efficiency. In fact, the MATLAB could be adapted to write the series code in other languages. Section 6 presents ODE solvers, fixed and adaptive step, that use the generated series recurrence code. Illustrative examples are provided with accuracy and efficiency comparison of different solvers, including `ode45`.

2. First examples and principles. All functions will be assumed to be real analytic in an open interval about an initial condition. So every function $y(t)$ and initial t_0 corresponds to a theoretically infinite array Y of Taylor series coefficients:

$$y(t) = Y[0] + Y[1](t-t_0) + Y[2](t-t_0)^2 + \cdots = \sum_{k=0}^{\infty} Y[k](t-t_0)^k.$$

For clarity of exposition and pseudo-code, we use index origin zero for the array Y and index into it with square brackets to emphasize the transformation from a function evaluation $y(t)$ to an array entry $Y[k]$. Any function about t_0 will correspond to an array, so a function u corresponds to array U.

For a preliminary example, consider Airy's Equation with initial values

$$y'' - t\,y = 0;\ y(t_0) = a;\ y'(t_0) = b.$$

As is done for numerical solution, use $y_1 = y$ and $y_2 = y'$ and rewrite the system $y_1' = y_2$ and $y_2' = t\,y_1$. In a programming environment, the user then defines the DE function $f(t,(y_1,y_2)) = (y_2, ty_1)$. In an algorithm to write a recurrence relation, each operation in f will corresponding to an auxiliary variable (function) assignment, here $u_1 = t\,y_1$, together with the system $y_1' = y_2$ and $y_2' = u_1$ that assigns one of the variables to each derivative, which we call a DE assignment. The variables and functions t, y_1, y_2, u_1 have corresponding arrays of series coefficients T, Y_1, Y_2, U_1. Initial values specify $Y_1[0] = a$ and $Y_2[0] = b$. Since $t = t_0 + (t - t_0)$, the array T is completely known, $T[0] = t_0$, $T[1] = 1$, and $T[k] = 0$ for $k \geq 2$. The U and Y arrays can be updated by comparing series terms in the assignments. Substituting series into the assignment $u_1 = t\,y_1$, yields

$$\sum_{k=0}^{\infty} U_1[k](t-t_0)^k = (T[0] + T[1](t-t_0))\,(\sum_{k=0}^{\infty} Y_1[k](t-t_0)^k),$$

so $U_1[k] = T[0]Y_1[k] + T[1]Y_1[k-1]$ for $k \geq 1$. Since $y_2' = \sum_{k=1}^{\infty} kY_2[k](t-t_0)^{k-1} = \sum_{k=0}^{\infty}(k+1)Y_2[k+1](t-t_0)^k$, the DE assignment $y_2' = u_1$ gives $(k+1)Y_2[k+1] = U_1[k]$. Similarly, $y_1' = y_2$, corresponds to $(k+1)Y_1[k+1] = Y_2[k]$. Starting from initial values, series coefficients for the solution of Airy's Equation can be generated from the following table.

auxiliary(s)	first steps	recurrence steps
$u_1 = t\, y_1$	$U_1[0] = T[0]Y_1[0]$	$U_1[k] = T[0]Y_1[k] + T[1]Y_1[k-1]$
DE assignments		
$y_1' = y_2$	$Y_1[1] = Y_2[0]$	$Y_1[k+1] = Y_2[k]/(k+1)$
$y_2' = u_1$	$Y_2[1] = U_1[0]$	$Y_2[k+1] = U_1[k]/(k+1)$

This shows the structure of our algorithmic Taylor method. To compare to classic series recurrence for $y = y_1$ in a DE course, use middle recurrence step above to form $Y_1[k+2] = Y_2[k+1]/(k+2)$, then substitute for this $Y_2[k+1]$ using the bottom step, to get $Y_1[k+2] = U_1[k]/((k+1)(k+2))$, and finally, by top step, $Y_1[k+2] = (t_0Y_1[k] + Y_1[k-1])/((k+1)(k+2))$. This agrees with the simpler cases found in [3], Section 5.2, where graphs and coefficient patterns are shown. For such simple DEs and assumptions, we could streamline derivation of simple recurrences, but our goal is to generalize and automate! Keeping arbitrary t_0 allows for the same recurrence program to work for each step of a piecewise series solution, using steps as in usual numerical solutions but giving polynomial approximations of the series solutions between relatively large steps.

A nonlinear example will motivate the two key principles behind recurrence relations for all other operations. This example

$$y' = \sin(y^2) \text{ with } y(t_0) = a \qquad \text{(Example 1)}$$

will guide exposition in subsequent sections. For the squaring operation, we consider the more general case of multiplication of two functions and corresponding multiplication of series. A product series converges as long as one of the original series is absolutely convergent (within the radius of convergence) and the other is convergent. To compute the kth coefficient of the product requires previously computed coefficients 0 to k for the multiplied functions, making a computer algorithm more practical than simple hand recurrence relations in most DE books. Specifically, suppose $w(t) = u(t)v(t)$, then

$$\sum_{k=0}^{\infty} W[k](t-t_0)^k = \left(\sum_{i=0}^{\infty} U[i](t-t_0)^i\right)\left(\sum_{j=0}^{\infty} V[j](t-t_0)^j\right).$$

For the multiplication to result in a specific power $(t-t_0)^k$, we need all i and j with $i + j = k$. This is called the Cauchy product.

THEOREM 2.1 (Cauchy Product).

If function	*then coefficient*
$w(t) = u(t)\, v(t)$	$W[k] = \sum_{i=0}^{k} U[i]\, V[k-i]$

.

For the general operation $\sin(u)$, we define two variables or auxiliary functions $s = \sin(u)$ and $c = \cos(u)$. Then $s' = c\,u'$ and $c' = -s\,u'$. With series adjusted for the derivative, this pair of multiplications can be computed by a Cauchy product. In general, suppose $w'(t) = \alpha\,u'(t)v(t)$ where α is a constant, so

$$\sum_{k=1}^{\infty} kW[k](t-t_0)^{k-1} = \alpha\left(\sum_{i=1}^{\infty} iU[i](t-t_0)^{i-1}\right)\left(\sum_{j=0}^{\infty} V[j](t-t_0)^{j}\right).$$

Comparing coefficients of $(t-t_0)^{k-1}$, we still need terms with $i+j=k$. The resulting principle uses previously computed coefficients 1 to k for u but only 0 to $k-1$ for v.

THEOREM 2.2 (Derivative Cauchy Product).

If function	*then coefficient for* $k \geq 1$
$w'(t) = \alpha\,u'(t)\,v(t)$	$W[k] = \frac{\alpha}{k}\sum_{i=1}^{k} iU[i]\,V[k-i]$

.

For the nonlinear ODE $y' = \sin(y^2)$, let $y_1 = y$ and initialize $Y_1[0] = y(t_0) = a$. Conceptually, evaluation of $\sin(y^2)$ generates a list of auxiliary variables, one for the operation y^2 and two for sin, followed by the DE assignment, as shown in the following table.

operation auxiliaries	derivative products
$u_1 = y_1\,y_1$	
$u_2 = \sin(u_1)$	$u_2' = u_3u_1'$
$u_3 = \cos(u_1)$	$u_3' = -u_2u_1'$
DE assignment	
$y_1' = u_2$	

This conceptual table corresponds directly to the initialization and recurrence steps given by Cauchy Products or Derivative Cauchy Products.

first steps	recurrence steps
$U_1[0] = Y_1[0]Y_1[0]$	$U_1[k] = \sum_{i=0}^{k} Y_1[i]\,Y_1[k-i]$
$U_2[0] = \sin(U_1[0])$	$U_2[k] = \frac{1}{k}\sum_{i=1}^{k} iU_1[i]\,U_3[k-i]$
$U_3[0] = \cos(U_1[0])$	$U_3[k] = \frac{-1}{k}\sum_{i=1}^{k} iU_1[i]\,U_2[k-i]$
DE assignment	
$Y_1[1] = U_2[0]$	$Y_1[k+1] = U_2[k]/(k+1)$

(2.1)

This process of creating recurrence steps generalizes to almost any ODE. Section 3 shows how most standard functions (like sin) have derivatives that can be written as just one-or-two products-or-quotients, called derivative products above. Recurrence for quotients will also follow from the Cauchy Product. A program can automate this process to take the DE function and directly write and/or implement code for the recurrence steps. Creating a conceptual table of recurrences by hand, as in (2.1) above, is a good exercise in learning this method and can be useful if you are primarily

interested in one differential equation and want to hand-code a specific solution. However, computer execution of code is preferred to generate the arbitrarily high series coefficients for the solution of the ODE about any t_0, given the initial value.

An alternative approach uses polynomial systems, as in [20]. For the above example, the derivative products could include $u_1' = 2yy' = 2y_1\,u_2$ and this could be substituted into the equations u_2' and u_3', yielding a polynomial system. Recurrence steps can be formed using only the Cauchy Product rule instead of specialized rules for each function, which may be pedagogically simpler. Theoretical error bounds have been derived using this method [23] [21] which can be generalized to our algorithms.

Returning to our recurrence steps, they are applied to produce the polynomial solutions shown in Figure 2.1 using $y(0) = 0.1$. One solution curve was generated by the usual MATLAB solver ode45. The recurrences generated coefficients for one Maclaurin polynomial of degree 25, also shown for degrees 5 and 10. The last curve of points uses the same recurrence steps to degree 12 but restarting with each marked point as chosen by an adaptive step size algorithm discussed in Section 6; accomplishing error within 10^{-9} using only 33 steps. In comparison, ode45 with requested tolerance of 10^{-9} used 119 steps.

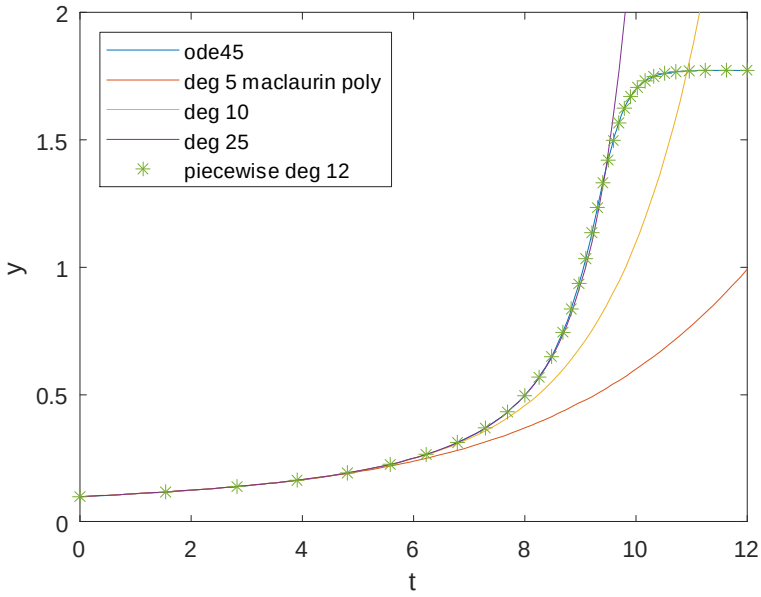

FIG. 2.1. *Using recurrence algorithm for solution of* $y' = \sin(y^2)$.

3. Recurrences for other operations and standard functions. Linear combinations are trivial. If $w(t) = \alpha\,u(t)$, then $W[k] = \alpha U[k]$. If $w(t) = u(t) + v(t)$, then $W[k] = U[k] + V[k]$. If $w(t) = u(t) + \alpha$, then, $W[0] = U[0] + \alpha$ and $W[k] = U[k]$ for $k \geq 1$.

Division is just a Cauchy Product solved for the quotient coefficient. If $w(t) = u(t)/v(t)$, then $u(t) = w(t)v(t)$, so by Cauchy Product $U[k] = \sum_{i=0}^{k} W[i]\,V[k-i] = W[k]V[0] + \sum_{i=0}^{k-1} W[i]\,V[k-i]$. Solving for $W[k]$ yields the following that uses W

only up to $k-1$.

If function	*then coefficient for* $k \ge 1$
$w(t) = u(t)/v(t)$	$W[k] = \Big(U[k] - \sum_{i=0}^{k-1} W[i]\, V[k-i]\Big)\, /V[0]$

(3.1)

Similarly, if $w'(t) = \alpha\, u'(t)/\, v(t)$, then $u'(t) = (1/\alpha)w'(t)v(t)$, so by Derivative Cauchy Product $U[k] = \frac{1/\alpha}{k} \sum_{i=1}^{k} iW[i]\, V[k-i]$. Then

$$\alpha U[k] = W[k]V[0] + \frac{1}{k} \sum_{i=1}^{k-1} iW[i]\, V[k-i].$$

Solving for $W[k]$ gives a principle that uses W and V only up to $k-1$.

If function	*then coefficient for* $k \ge 1$
$w'(t) = \alpha\, u'(t)/v(t)$	$W[k] = \Big(\alpha U[k] - \frac{1}{k} \sum_{i=1}^{k-1} iW[i]\, V[k-i]\Big)\, /V[0]$

For standard functions, we just need to find derivative product (or quotient) relationships, as was done for sin in the previous section. Most are shown in

operation	product/quotient	recurrence for $k \ge 1$
$w = \exp(u)$	$w' = u'w$	$W[k] = \frac{1}{k} \sum_{i=1}^{k} iU[i]\, W[k-i]$
$w = \ln(u)$	$w' = u'/u$	$W[k] = \Big(U[k] - \frac{1}{k} \sum_{i=1}^{k-1} iW[i]\, U[k-i]\Big)\, /U[0]$
$w = \sqrt{u}$	$w' = (1/2)u'/w$	$W[k] = \Big(\frac{1}{2}U[k] - \frac{1}{k} \sum_{i=1}^{k-1} iW[i]\, W[k-i]\Big)\, /W[0]$ †
$w_1 = u^\alpha$	$w_1' = u'w_2$	$W_1[k] = \frac{1}{k} \sum_{i=1}^{k} iU[i]\, W_2[k-i]$
$w_2 = \alpha\, u^{\alpha-1}$	$w_2' = {}_{(\alpha-1)}w_1'/u$	$W_2[k] = \Big({}_{(\alpha-1)}W_1[k] - \frac{1}{k} \sum_{i=1}^{k-1} iW_2[i]\, U[k-i]\Big)\, /U[0]$
$w_1 = \sin(u)$	$w_1' = u'w_2$	$W_1[k] = \frac{1}{k} \sum_{i=1}^{k} iU[i]\, W_2[k-i]$
$w_2 = \cos(u)$	$w_2' = -u'w_1$	$W_2[k] = \frac{-1}{k} \sum_{i=1}^{k} iU[i]\, W_1[k-i]$
$w_1 = \tan(u)$	$w_1' = u'w_2$	$W_1[k] = \frac{1}{k} \sum_{i=1}^{k} iU[i]\, W_2[k-i]$
$w_2 = 1 + w_1^2$	$w_2' = 2w_1'w_1$	$W_2[k] = \frac{2}{k} \sum_{i=1}^{k} iW_1[i]\, W_1[k-i]$ ‡
$w_1 = \sin^{-1}(u)$	$w_1' = u'/w_2$	$W_1[k] = \Big(U[k] - \frac{1}{k} \sum_{i=1}^{k-1} iW_1[i]\, W_2[k-i]\Big)\, /W_2[0]$
$w_2 = \cos(w_1)$	$w_2' = -w_1'u$	$W_2[k] = \frac{-1}{k} \sum_{i=1}^{k} iW_1[i]\, U[k-i]$
$w_1 = \tan^{-1}(u)$	$w_1' = u'/w_2$	$W_1[k] = \Big(U[k] - \frac{1}{k} \sum_{i=1}^{k-1} iW_1[i]\, W_2[k-i]\Big)\, /W_2[0]$
$w_2 = 1 + u^2$	$w_2' = 2u'u$	$W_2[k] = \frac{2}{k} \sum_{i=1}^{k} iU[i]\, U[k-i]$ ‡

TABLE 3.1
Recurrences for many standard functions

in Table 3.1. Some of these can be simplified as shown below, when one array is in Derivative Cauchy Product with itself.

$$(\dagger)\quad \frac{1}{k} \sum_{i=1}^{k-1} iW[i]\, W[k-i] = \frac{1}{2} \sum_{i=1}^{k-1} W[i]\, W[k-i]$$

$$(\ddagger)\quad \frac{1}{k} \sum_{i=1}^{k} iW[i]\, W[k-i] = \frac{1}{2} \sum_{i=0}^{k} W[i]\, W[k-i]$$

To derive these, observe that replacing $iW[i]\,W[k-i]$ with $(k-i)W[i]\,W[k-i]$ does not change the sum in range 1 to $k-1$ or range 0 to k, and adding both sums gives the simplification. Moreover, the range could then be cut in half by using symmetry, though even and odd cases must be handled differently. Which version is more efficient can depend on the specific implementation and computational environment. One option is to store $kW[k]$ beside $W[k]$, sometimes requiring no new computation since $\frac{1}{k}$ is often the last step in computation of $W[k]$. A small disadvantage of (‡) is that it increases the summation range by adding $i = 0$, which is significant in the multivariable generalization of these recurrences [16]. (For consistency, every recurrence in Table 3.1 comes from a derivative product or quotient, but alternative derivations of recurrences with (†) or (‡) follow from directly forming a square without a derivative, as in $w = \sqrt{u}$ is $u = w^2$, and solving for $W[k]$.)

Many standard functions not in Table 3.1 have similar recurrences that require at most two Cauchy Products (including Derivative Cauchy Products). A particularly worthwhile addition is for $w_1 = \cot(u)$ which is analogous to tan, only inserting a negation. Such minor analogous changes produce recurrences for most other inverse trigonometric and hyperbolic trigonometric or inverse hyperbolic trigonometric functions. However, any such function referring to sec or csc seems to require three Cauchy Products. Hence, it is just as efficient to implement them as in $\sec(u) = 1/\cos(u)$ or $\operatorname{csch}^{-1}(u) = \sinh^{-1}(1/u)$, requiring three Cauchy Products including division as one. In fact, Table 3.1 suffices to efficiently write many common functions, such as $\cos^{-1} u = (\pi/2) - \sin^{-1} u$ or $\sinh u = (e^u - e^{-u})/2$, requiring two Cauchy Products. Such identities can be called by the software that automates producing recurrences, so that a user may use any of these functions in the original DE function.

Implementing power should use several cases. The Table 3.1 formula for constant exponent requires a nonzero base and uses two Derivative Cauchy Products. For exponents 2, 3, and 4, repeated multiplication (or squaring) is at least as efficient and allows zero base. Exponent $1/2$ is square root and -1 is a simple division. For variable exponent, using $v^u = \exp(u\ln(v))$ is a good implementation where positive base is needed and the general case requires three Cauchy products. If base v is constant, the scalar multiple results in only two Cauchy products.

4. Generated code for power series solution. The conceptual process of the examples in Section 2 is implemented in a MATLAB program `makepsmcode.m` that takes a DE and returns a MATLAB program that generates series coefficients for the series solution. We first look at the use and output of `makepsmcode`, rather than the programming behind the function. This perspective can be useful in a differential equations or numerical analysis course where the focus is on the usefulness of Taylor series solutions. It may be enough to explain that the programming behind the function must interpret the DE file, either by operator overloading (as in explained in Section 5) or by a text file parser, and write the output file of recurrence steps in response.

The input to `makepsmcode` is what we will call a **DE file**, as used in numerical solutions. Example 1, $y' = \sin(y^2)$, could be given in the DE file `fex1.m` with the following contents.

```
function dydt = fex1(t,y)
dydt = sin(y^2);
```

We expect and implement only scalar functions (so `^` not `.^` in MATLAB). Variable names (`t`, `y`, `dydt`) could be anything but the returned series program will refer to independent `t` and function `y`. As is usual in numerical solutions, any DE is written

as a system of first order DEs where `y` can be a column vector function. The only other input to `makepsmcode` is the number of DEs in the system, which is the length of the vector function `y` (which is 1 in this simple first example).

The command `makepsmcode(@fex1,1)` writes a new file `fex1series.m` with the following contents. MATLAB code comments use the percentage sign `%` and execution ignores the remainder of line.

```
function coefs = fex1series(t0, y0, deg)
%FEX1SERIES finds series coefs for soln of y' = f(t,y) about t0 to deg
% Where f(t,y) was given by:
%    function dydt = fex1(t,y)
%    dydt = sin(y^2);
% Automatically generated by makepsmcode.m, 04-Jun-2024 12:00:40.

numfops = 3;  % number of scalar operations used to evaluate f
numys = length(y0);
T = [t0,1,zeros(1,deg-1)];
Y = zeros(numys,deg+1);
U = zeros(numfops,deg+1);
if isa(t0,'sym') || isa(y0,'sym')   % allow symbolic, including vpa
    T = sym(T); Y = sym(Y); U = sym(U);
end
Y(:,1) = y0;

% Evaluate f(t0,y0) recording value after each scalar operation
U(1,1) = Y(1,1) * Y(1,1);
U(2,1) = sin(U(1,1));
U(3,1) = cos(U(1,1));
% Update Y linear coefficient by y' = f(t,y)
Y(1,2) = U(2,1);

% Now recurrence rules for each operation in evaluation of f
for j = 2:deg   % j-1 is the power of term at index position j
    U(1,j) = Y(1,j:-1:1) * Y(1,1:j).';
    tempprime = ( U(1,2:j) .* (1:(j-1)) ).';
    U(2,j) = (U(3,(j-1):-1:1) * tempprime)/(j-1);
    U(3,j) = -(U(2,(j-1):-1:1) * tempprime)/(j-1);
    % Update Y next coefficient using y' = f(t,y)
    Y(1,j+1) = U(2,j)/j;
end

coefs = Y;
end
```

A number of the implementation choices are specific to MATLAB, where array index origin one is a hassle for translating series index. (This hassle that could be avoided in languages such as Python and C++ with index origin zero.) The arrays $U_i[k]$ and $Y_1[k]$ in (2.1) are implemented in two-dimensional arrays using `U(i,k+1)` and `Y(1,k+1)` respectively. Let's explain the more complicated computation of `U(2,j)`,

starting with the formula from (2.1) and simply changing the array representation, leaving everything else the same:

$$U_2[k] = \frac{1}{k}\sum_{i=1}^{k} iU_1[i]\,U_3[k-i] \quad \text{in index origin zero becomes}$$

$$\mathtt{U(2,k+1)} = \frac{1}{k}\sum_{i=1}^{k} i\mathtt{U(1,i+1)\,U(3,k-i+1)} \text{ in index origin one, or}$$

$$\mathtt{U(2,j)} = \frac{1}{(j-1)}\sum_{i=1}^{j-1} i\mathtt{U(1,i+1)\,U(3,j-i)} \text{ using } \mathtt{j} = k+1.$$

In any language, the key to efficient series computation is efficient implementation of such Cauchy product sums. In MATLAB, we compute with vectors of values (instead of a loop on i, or $m = i+1$). We use three vectors for values in the summand, using the MATLAB colon operator: `(1:j-1)`, `U(1,2:j)`, and `U(3,j-1:-1:1)` where the order is efficiently reversed by `:-1:`. We multiply the components of the first two using `(1:j-1).*U(1,2:j)`, resulting in coefficients corresponding to u_1'. We need the dot product of this u_1' vector with `U(3,j-1:-1:1)`. Timings showed that a dot product (the key to any Cauchy product) is most efficiently done by matrix multiplication in MATLAB, so one vector times the transpose of the other. We transpose the u_1' vector into a reusable column vector, leading to the two lines in `fex1series`.

```
tempprime = ( U(1,2:j) .* (1:(j-1)) ).';
U(2,j) = (U(3,(j-1):-1:1) * tempprime)/(j-1);
```

Implementing other lines of (2.1) is an exercise in similar techniques. These programming details can be confusing and error-prone by hand. By trusting the programmer of `makepsmcode`, a user can understand that lines of the recurrence loop accurately reflect the steps in (2.1) and fruitfully use and appreciate the automatically generated code, without concern for the indexing details.

The output of `makepsmcode`, the function `fex1series` in this example, may be used directly or called by an ODE solver program. The run `fex1series(0,0.1,25)` returns a row vector of coefficients of the Maclaurin series solution, which we assign to variable `coefs`, starting with constant term `0.1` through degree 25 term coefficient `8.6950e-27`. Each coefficient can be observed to be about one-tenth of the previous term, showing radius of convergence around 10. One may evaluate the degree 25 polynomial using the MATLAB function `polyval` or by programming Horner's rule, as is done in `serieseval` in the package of functions that accompany this paper. Such evaluations gave the polynomial plots in Figure 2.1. An example value at $t = 8.0$ is `0.4972` returned by `polyval(flip(coefs),8.0)`. One may also compute symbolic series solutions if the MATLAB installation includes the symbolic toolbox. Using the `if` statement in the above program, `coefs = fex1series(0,sym('a'),3)` returns `[a, sin(a^2), a*cos(a^2)*sin(a^2), (cos(a^2)*(2*sin(a^2)^2 + 4*a^2*cos(a^2)*sin(a^2)))/6 - (2*a^2*sin(a^2)^3)/3]`.

For a system of ODES, the series program returns a matrix of coefs with one row for each function in the system solution. For example, the second order equation $y'' = \sin(y^2)$ uses $y_1 = y$ and $y_2 = y'$ to make the system $y_1' = y_2$ and $y_2' = \sin(y_1^2)$. The DE file defining this system is

```
function dydt = fex1b(t,y)
dydt = [y(2); sin(y(1)^2)];
```

The input `y` or `y0` to this function is a column vector. Avoid preallocating vector output in the `fex1b.m` file. (Preallocation using `dydt = zeros(2,1)` leads to an error when the code behind `makepsmcode` overloads `y` as explained in Section 5. Then, in MATLAB, `dydt(1) = y(2)` would attempt to convert object `y(2)` into a double, when we need `dydt` to be a vector of objects. Concatenating objects works fine or one could preallocate using `dydt = y`.)

For this second order system, the program `fex1bseries` written by `makepsmcode` is almost the same as `fex1series` above. Only the "Update Y" lines are changed: replacing linear coefficient assignment `Y(1,2) = U(2,1);` with the two lines `Y(1,2) = Y(2,1); Y(2,2) = U(2,1);` and recurrence assignment `Y(1,j+1) = U(2,j)/j;` with two lines `Y(1,j+1) = Y(2,j)/j; Y(2,j+1) = U(2,j)/j;`. The output of `fex1bseries` is a matrix with two rows of series coefficients, for y_1 and y_2 respectively. In this case, the second series is just the derivative of the first, which is how `Y(1,j+1)` is updated. The Horner's rule in `serieseval` can easily evaluate both polynomials by doing computations with columns. Unfortunately, `polyval` does not work on a matrix of coefficients. Of course, this second order differential equation has a significantly different solution than Example 1.

5. Automatic code writing by operator overloading. We now explain how we use operator overloading to write code for the Taylor series solution of an ODE. We hope some of these concepts and technical details will be interesting, as object-oriented programming or MATLAB software, and helpful for developing such software. The general idea is that evaluation of a DE file for `f(t,y)` will do more than just compute numerical value(s), as it would if t and y were doubles or y was an array of doubles. When t and y are objects, or y is an array of objects, of a special class that you define, then each operation in evaluating `f(t,y)` will call the method for that operation (an overloaded operation) in the class definition file. Using a `trace` class, each operation in one evaluation of `f(t,y)` writes text string(s) for the evaluation step and for the recurrence step.

In the example of the previous Section, the operation `y^2` in `fex1`, with `y` being a `trace` object, produces the text lines for the operation result `U(1,1)` and for the corresponding series coefficient recurrence `U(1,j)`, as seen in `fex1series`. But the trace class has to know that this is the first operation 1, and when sin is called we do operation 2 and also an operation 3 for cos. This counting is an important and tricky process enabled by a `Static` class method `count_val` that has `persistent` local variables `count`, `vallist`, `oplist`, and `reclist`. The class is tracing through the computation of `sin(y^2)`where each needed operation calls `count_val` to increment `count`, append new value to `vallist`, append the text line(s) for the operation to `oplist`, and append the text line(s) for the recurrence to `reclist`.

Each starting and intermediate value in the calculation is an instance of a `trace` object where the class definition file `trace.m` begins as follows. In this Section, some code comment lines are stripped out for condensed presentation.

```
classdef trace
   properties
      val  % expression value after its operation/function
      opcount % index of val in vallist maintained by count_val
      tlinear % T/F: operations to this point are equivalent to at+b
   end
```

The code comments and above paragraph explain the meaning of `val` and `opcount` in ongoing evaluations. Boolean property `tlinear` is not necessary but, if true, allows for writing more efficient code when using this object in a Cauchy product, since it has only two nonzero terms in the corresponding series coefficients. Variables are also trace objects: `t` is initialized with t_0 as `val` and `opcount` 0, flagging the independent variable. Variable `y` is initialized as an array of `trace` objects, where `y(i)` has `val` from the corresponding entry in y_0 vector and `opcount` $-i$, where negative flags the ith component of `y`. A positive `opcount` is assigned by `count_val` and indicates that the object resulted from an operation in `f(t,y)` and corresponding code should write the array name `U` with that `opcount` as first index. In executing overloaded `f(t,y)`, any reference to the variables or to any intermediate variable values (in trace objects) will have access to the `opcount`s showing what name and what index value is to be used in writing the code. The user's DE file for `f(t,y)` does not have to be just one expression in `t` and `y` but can actually use any variable names and any number lines and local variables. Our implementation code, will use `t` and write `T` for the first argument, `y` and `Y` for the second argument, and `u` and `U` for all intermediate step values whether explicitly assigned a name by the user's code or not. Optionally, we use `Static` functions `name` and `nextname` to facilitate writing the correct variable names and indices, as explained further below.

The driver program `makepsmcode` initializes the objects, calls the overloaded function, and gets the `count_val` accumulations, for use in writing the code. The necessary starting lines of `makepsmcode` follow. We don't show the code that makes random values for `t0` and `y0`, if those arguments are omitted in the call.

```
function newfunc = makepsmcode(fhandle, numDEs, t0, y0)
clear trace  % restart trace class, clears all persistent variables
t = trace(t0, 0);  % 0 indicates independent variable
y = trace(y0, 1);  % 1 indicates dependent variable in DE for y(t)
h = feval(fhandle, t, y);  % overloaded evaluation of f(t,y)
[fvalue, location] = valueandopcount(h);  % vectors for f(t,y) results
[numops, valuelist, oplist, recurlist] = trace.count_val;
```

Here we see use of the class constructor `trace` to initialize the objects as described in the paragraph about `val` and `opcount`. From the resulting object `h`, the `opcount` (returned in `location`) gives the index of the `U` used when writing the `"% Update Y"` step(s) for DE assignment(s) such as $y\prime = h$. For example, in the run of `makepsmcode` that produced `fex1series`, the object `h` had `val` $\sin(y_0^2)$ (not used by `makepsmcode`) and `opcount` 2 which is used to write the Update Y line `Y(1,j+1) = U(2,j)/j;`. If `y` is a vector, then `h` will also be a vector of objects and `valueandopcount` returns vectors of doubles. When `trace.count_val` is called without any arguments, it simply returns the accumulated persistent variables. Output `oplist` and `recurlist` are cell arrays with each cell containing a string for a line of code to be written into an output file. In creation of `fex1series`, they contain the lines of code for all `U` assignments, 3 lines in `oplist` and 4 in `recurlist`. Immediately after the seven lines of `makepsmcode` above, the next lines augment `oplist` and `recurlist` with code for the Update Y steps as just mentioned. The remainder of `makepsmcode` is the straight-forward, unattractive code to write all of the series code into an appropriately named file. The example output `fex1series` shows the overall structure (in addition to `oplist` and `recurlist`) that remains the same for all applications.

Let's turn to the coding of methods inside the trace class for each of the overloaded operations (either an arithmetic operation or standard transcendental function). Specifically, we look at overloading simple negation and overloading `sin` as typical examples. MATLAB Help has a page on "Operator Overloading" that includes a table of operators (using usual symbols) and the corresponding method name that you can overload. Simple negation `-a` has method name `uminus` (for unary minus). Also note that scalar multiplication `a*b` is actually called `mtimes`, since matrix multiplication is the default in MATLAB. Overloading `uminus` is extremely simple but shows the object handling steps that appear in all the overloaded operations.

```
function h = uminus(u)
   %trace/UMINUS overloads negation with a trace object argument
   % h = -u  for both operation value and series recurrence code
   newval = -u.val;
   h = trace(newval);
   opstr  = [trace.nextname,',1) = -', trace.name(u.opcount),',1);'];
   recstr = [trace.nextname,',j) = -', trace.name(u.opcount),',j);'];
   h.opcount = trace.count_val(newval, {opstr}, {recstr});
   h.tlinear = u.tlinear;
end
```

An example call to `uminus` might be in the expression `-sin(y^2)`, adding just one negation to our `fex1` example. Operations 1, 2, and 3 are ^2, sin, and cos, as discussed in the previous section, and the persistent class variable `count` will be 3 when `uminus` called. The call to `sin` will return a trace object which becomes the actual parameter input for formal parameter `u`. The `uminus` call creates a new trace object, locally called `h`, which will be returned as the output of `uminus`. The class constructor `trace` sets the `val` property of `h`, but other properties need to be updated. In our example call, `opstr` would become the text string `'U(4,1) = -U(2,1);'` where class static method `trace.name(u.opcount)` would return string `'U(2'` and `trace.nextname` would return `'U(4'`. In different example calls `-t` and `-y(2)`, `u.opcount` values 0 and -2 would produce `'T(1'` and `'Y(2'`, respectively. Method `nextname` uses `trace.count_val` without any argument to get `count`. However, `count` is not incremented until `trace.count_val` is given the next items to append to the persistent lists, as on the second to last line of `uminus`. Notice that `opstr` and `recstr` are put into a cell by using braces, allowing different length strings to be appended to the vertical array of cells `oplist`.

In `makepsmcode`, we use other cases of constructor `trace`, where 0 and 1 flags initialize all the properties. Additionally, `trace` with no arguments will create a completely empty object, a case that is used in implicit construction of arrays of objects.

With `uminus` providing a template for handling the objects, strings, and lists, the overloading of other functions primarily involves creating strings for the recurrence relations in Table 3.1. When a pair of recurrences are required (needing w_1 and w_2 in the table), the code will define two objects and append both associated persistent list items, but usually only the first (requested) object is returned. In the example of sin and cos, both functions could be useful, so we define a method below that returns both objects.

```
function [s, c] = sincos(u)
%trace/SINCOS returns trace objects for sin(u) and cos(u)
% s = sin(u) and c = cos(u)
% s' = c * u' and c' = -s * u' Cauchy products series recurrence
newval = sin(u.val);
s = trace(newval);
sname = trace.nextname;    % 'U(i' where i is trace.count_val + 1
cname = trace.nextname(2); % 'U(i' where i is trace.count_val + 2
uname = trace.name(u.opcount);
opstr = [sname, ',1) = sin(', uname,',1));'];
if u.tlinear
   reccell = {[sname,',j) = ',cname,',j-1)*',uname,',2)/(j-1);']};
else
   % tempprime pulls coefs for u' out of array coefs for u
   recstr1 = ['tempprime = ( ',uname,',2:j) .* (1:(j-1)) ).'';'];
   recstr2=[sname,',j) = (',cname,',(j-1):-1:1) * tempprime)/(j-1);'];
   reccell = {recstr1;recstr2};
end
s.opcount = trace.count_val(newval, {opstr}, reccell);
s.tlinear = false;

newval = cos(u.val);
c = trace(newval);
opstr = [cname, ',1) = cos(', uname,',1));'];
if u.tlinear
   recstr = [cname,',j) = -',sname,',j-1)*',uname,',2)/(j-1);'];
else
   recstr=[cname,',j) = -(',sname,',(j-1):-1:1) * tempprime)/(j-1);'];
end
c.opcount = trace.count_val(newval, {opstr}, {recstr});
c.tlinear = false;
end
```

In the general case, `u.tlinear` is false and evaluation appends two strings using `opstr` twice and three strings using `reccell` and `recstr`. This produces the five lines in `fex1series` for `U(2,1)`, `U(3,1)`, `tempprime`, `U(2,j)`, and `U(3,j)`, in Section 4. We see how `u.tlinear` will simplify the code if the argument `u` is linear in t, as in example `sin(3*t+1)`, where `tempprime` would have only one nonzero term. For other operations, such as multiplication, `u.tlinear` could differently handle the case of a Cauchy product with only two nonzero terms (as in the preliminary example of Section 2), although we do not implement it since little time improvement was noticed in MATLAB.

The overloaded function `sin` simply calls `sincos` and returns only the first object output.

```
function h = sin(u)
   %trace/SIN overloads sine with a trace object argument
   [h, g] = sincos(u);
end
```

Similarly, `cos` returns `g`. This means that if a DE file uses both sin and cos with

the same argument, then the code will be duplicated unnecessarily in output series file. One way around this is to make a version of the DE file that calls sincos on a separate line and uses the output variables, but this version of the DE file will no longer work with doubles as in ordinary numerical computation. One general lesson is that a more efficient DE file will make a more efficient series code, so things like common expressions should be computed and saved to a variable on a separate line. We can overload other functions that are defined in terms of just the few given in Table 3.1, as in the following.

```
function h = csc(u)
   %trace/CSC overloads csc of a trace object argument
   h = 1/sin(u);
end
```

Using `csc` will result in code for three Cauchy products by calling `sincos` indirectly and `mrdivide` which overloads the division with one Cauchy product defined by (3.1). This brings up another issue in overloading many functions: there are simpler special cases.

When overloading a function with two arguments, including `plus`, `minus`, `mtimes`, `mrdivide`, and `mpower`, we separately program the case when one of the arguments is not a trace object but is numerical (as in 1/sin(u) above). In defining trace method `mtimes(u,v)`, we have three cases (`~` is boolean operator not): `~isa(u,'trace')`, `~isa(v,'trace')`, and else general case which is the only one that requires a Cauchy product. Note that if both arguments are numerical, the overloaded trace function will not be called but the numerical value is computed as usual. For example `p=3; dydt = 2*p*y;`, will show up in series code same as `6*y` would. We use `num2str(u,16)` to write the double value with up to 16 digits if needed. Method `mpower(u,r)` uses several special cases as discussed at the end of Section 3. We've frequently used `u^2`, where `mpower` simply calls `u*u`, and thus `mtimes`, in this case. If `r` is variable, so `isa(r,'trace')`, then `mpower` calls `exp(r*log(u))`, calling three overloaded functions that result in three Cauchy products, unless `u` is numerical and then only two overloaded functions are called. The only Cauchy product directly programmed in `mpower` is when `r` is a non-special numerical constant that uses the u^a pair of recurrences in Table 3.1.

As alternatives to the implementation of this paper, we have programmed two other approaches to using object-oriented computing for an algorithmic Taylor series solution of ODEs. The simplest approach is found in [15] where any function u is an instance of a **series** class that holds the vector of series coefficients to some degree d. Operations on series objects will perform the associated recurrence in Section 3. For a function of $f(t)$, the series object for t is known to degree d, so one overloaded evaluation of f will return the series coefficients for $f(t)$. This implementation is recommended for finding Taylor series for function expressions. However, it is inefficient in solving $y' = f(t, y)$. If coefficients for y are known to only degree k (initially $k = 0$), then overloaded evaluation of $f(t, y)$ will return the coefficients for y' which gives the next coefficient of degree $k + 1$. The overloaded evaluation must happen inside a loop on degree k, which does wasteful recalculation of all the lower order series coefficients for y and all the intermediate operations that are not saved anywhere. The second alternative approach saves these intermediate series values for a class that performs the overloaded operations instead of writing them. This implementation used two classes for overloading, **valtrace** and **sertrace**. The first evaluation of $f(t, y)$ uses valtrace objects much as in trace, but counting and

storing (in vallist) only the numerical value at each step (nothing about higher series terms). The sertrace class can be used in the looping just described for the first alternative but without the wasted recalculation. A persistent handle (or pointer) to one array `uall` of sertrace objects, one for each of the steps (enumerated by valtrace), plays the role of the array `U` in code written by `makepsmcode`. Evaluation of $f(t, y)$ is done inside a loop on degree k, where each overloaded operation computes the next coefficient using values in `uall`. This performs exactly the same floating point computations as would code generated by `makepsmcode`. However, it is slower due to the overhead of all the calls and references. The generated recurrence code would be even more efficient in a compiled language.

6. DE solver using high-order Taylor series. Generated code for series recurrence can easily be used for numerically solving the associated differential equation using a high-order Taylor series method. The step size through time can be fixed or variable as in most numerical solvers. The order of the solution method (the degree to which the series is computed) can also be chosen by the user, chosen by the program, or variable along with step size. We begin with a simple fixed step size `h` example.

The program `odepsmh` is an explicit numerical solver that repeatedly uses (t_k, y_k) to compute (t_{k+1}, y_{k+1}) (as in Euler or RK4 methods). The next value y_{k+1} is computed from the Taylor series about t_k evaluated at t_{k+1}. For a DE file, `makepsmcode` need be run only once. This solver will expect or create series recurrence code, say `fex1series.m`, to match the DE file, respectively `fex1.m`, given by input function handle. Note: if you change a DE file contents, you must delete the corresponding series file, so the correct one will be created. The series recurrence code is given the starting value of (t_k, y_k) and degree `deg`, and returns series coefficients Y from 0 to `deg`. The polynomial evaluation $y_{k+1} = \sum_{j=0}^{\deg} Y[j](t_{k+1} - t_k)^j$ is most efficiently done by Horner's rule, done by calling `serieseval` or implementation in `odepsmh` code below. Each y_k can be a column vector, so we use array `y` where each column `y(:,k)` are values of all components of y at one time. The series coefficients are held in matrix `coefs` which is the `Y` matrix computed by series recurrence code, so that `coefs(:,m)` holds coefficients for all components of degree $m - 1$. Index origin one requires the offset index from 1 to deg + 1. We evaluate all components of y in parallel using column arithmetic.

```
function [t,y] = odepsmh(fhandle,tspan,h,y0,deg)
t0 = tspan(1);
tend = tspan(2);
if h*(tend - t0) < 0; h = -h; end  % match sign of h to direction
t = (t0:h:tend).'; % t is a column vector of times t(k), with t(1)=t0
if t(end) ~= tend
    t = [t;tend];
end
numpts = length(t);

% series recurrence code is generated if it doesn't already exist
seriesfuncname = [func2str(fhandle),'series'];
if isfile([seriesfuncname,'.m'])
    fseries = str2func(seriesfuncname);  % return function handle
else
    fseries = makepsmcode(fhandle,length(y0));
```

```
end

y = zeros(length(y0),numpts); % index by y(component, time)
y(:,1) = y0;
for k = 1:(numpts-1)
    coefs = fseries(t(k),y(:,k),deg);

    % horners rule to evaluate polynomial in powers of (t(k+1)-t(k))
    val = coefs(:,deg+1);
    h = t(k+1)-t(k); % could differ on last step
    for m = deg:-1:1
        val = val*h + coefs(:,m);
    end
    y(:,k+1) = val;
end
y = y.'; % transpose into output form
```

We test `odepsmh` on another example of a forced damped pendulum model that can be chaotic. The equation

$$y'' = -\sin(y) - 0.1y' + \cos(t)$$

is given in the DE file:

```
function dydt = fdpendulum(t,y)
dydt = [ y(2); -sin(y(1)) - 0.1*y(2) + cos(t)];
```

We start the pendulum at the bottom and give it a strong kick using $y(0) = 0, y'(0) = 2$. The call `[tpsm,ypsm] = odepsmh(@fdpendulum,[0,200],0.6,[0;2],20)` uses large step size $h = 0.6$ and degree 20 series to compute an accurate solution through time 200. A phase plane plot of the solution is shown in red in Figure 6.1. We com-

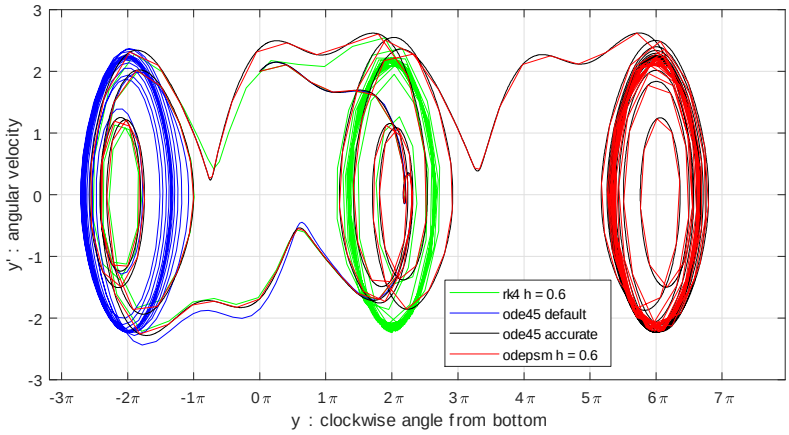

FIG. 6.1. *Different numerical solutions of a forced damped pendulum*

pare three other numerical solutions that show the sensitivity. The phase plane shows different "wells" around even multiples of π where the pendulum can oscillate until it goes over the top (with positive or negative velocity) into a different well. The blue

curve shows that a call to MATLAB's `ode45(@fdpendulum,[0,200],[0;2])` with default options does not find the correct well. Neither does the standard Runge-Kutta order 4 with fixed step size of $h = 0.6$. The black accurate curve was generated by ode45 using `options = odeset('RelTol',2.3e-14,'AbsTol',1e-15,'Refine',1)`. The red chords cut across the black curves, not because of inaccuracy but because only the large step size points are computed. Every red curve chord plots just endpoints of a 0.6 span of a the degree 20 Taylor polynomial computed by the odepsmh code.

The following table shows efficiency and accuracy of the comparative numerical solution runs, with the first four rows being the runs reflected in Figure 6.1.

	num steps	mean step	y(200) value	error	sec
runge kutta 4:	334	0.6000	4.85351621078251	1.3e+01	6.7e-04
ode45 default:	256	0.7813	-7.71282407459629	2.5e+01	2.3e-03
ode45 accurate:	30989	0.0065	17.41704528241850	2.2e-12	5.2e-02
odepsmh deg 20:	334	0.6000	17.41704249607110	2.8e-06	2.1e-02
odepsmh deg 20:	800	0.2500	17.41704528241632	0.0e+00	4.9e-02
odepsmJZ deg 16:	794	0.2519	17.41704528241750	1.2e-12	4.6e-02
odepsmJZ deg 5:	690	0.2899	17.42184618980130	4.8e-03	1.6e-02

With only 334 steps, `odepsmh` has absolute error on the order of 10^{-6}. The first two rows show unsuccessful results despite a similar number of steps. Error of 10^{-12} was found by ode45 using about a hundred times the number of steps. (These step counts adjust for the ode45 default refinement factor of 4, which returns four points for each step of the solver. Thus 1024 points were plotted for the blue curve. With `options`, given above for the accurate call, ode45 performed 30989 steps and returned just those endpoints.) The last three rows show other uses of the generated `fdpendulumseries` code. On the fifth row, experimentation showed that 16 significant digits of accuracy were obtained using 800 steps of step size $h = 0.25$ and degree 20 Taylor polynomials, so this end value was used as the true value. The sixth and seventh rows show use of an adaptive method `odepsmJZ` described in the next paragraph. Program `odepsmJZ`, called with tolerance of `1e-13`, chose to use degree 16 Taylor polynomials and 794 steps, which was similar to the row above and had comparable performance to the accurate ode45 call. The last row succeeds in finding enough accuracy for a reliable plot (where ode45 default fails), by calling odepsmJZ with tolerance `1e-3` which used degree 5 Taylor polynomials and 690 steps. Timings in the last column are averaged over 100 runs. Timings for the odepsm programs do not include the time to generate `fdpendulumseries`, since this is needed only once for a DE file. One isolated run of odepsmh with $h = 0.6$, including generating the series code, took `4.1e-1` seconds, an order of magnitude longer. However, ode45 accurate also took `2.1e-1` seconds on its first call from a new session of MATLAB. Clearly, repeated calls can run more rapidly. Of course, computer hardware, software, and status affect timings. (We used an old Windows 10 laptop running MATLAB 2024a.) While exact timings vary, the relative performance of repeated runs, shown in the above table, seems to hold. For this example of a forced damped pendulum, when looking for the best accuracy, odepsmh can beat ode45 with far fewer steps and comparable or better time.

When using series recurrence code in an adaptive solver, there are more ways to adjust based on error tolerance than in standard adaptive programs. In addition to adjusting step size, we can adjust the order of series terms to be generated, either

once using tolerance or dynamically based on error estimates. Also, series coefficients can be used to estimate error and/or estimate radius of convergence that informs step size choice. In `odepsmJZ`, we implemented a version of the adaptive algorithms used by Jorba and Zou in [12].

First, we sketch the principles used from [19] that optimize efficiency within error tolerance ε, thinking asymptotically and ignoring constants. On each step, compute series terms to degree N, estimate the radius of convergence ρ, and choose step size h as a fraction (less than half) of ρ. Assuming the series is asymptotically geometric, next value $y_{k+1} = \sum_{j=0}^{N} Y[j](t_{k+1} - t_k)^j$ has error that is bounded by the last term that is roughly equivalent to $\rho^{-N}h^N$. Set this equal to ε and solve for $N = \ln(\varepsilon)/\ln(h/\rho)$. The cost of computing the series coefficients on each step is assumed to be proportional to the cost N^2 of one Cauchy product. The number of steps is proportional to $1/h$, so we wish to minimize N^2/h, or maximize h/N^2. Substituting for N in terms of h, it suffices to maximize $h\,(\ln(h/\rho))^2$. Calculus finds the maximum at $h = \rho/e^2$, fixing the fraction of ρ. Thus $N = -\ln(\varepsilon)/2$ depends only on the tolerance.

Our adaptive code follows [12] with practical adjustments to this theoretical N and h. Assume, for simplicity, that one variable `tolerance` is used to request the same absolute and relative errors. The implemented N is one more than theory and uses the ceiling function:

```
deg = ceil(1-log(tolerance)/2).
```

So, requested tolerances `1e-13` and `1e-3` use degrees 16 and 5, respectively. Our program starts with the header

```
function [t,y,deg] = odepsmJZ(fhandle,tspan,y0,tolerance)
```

and has much the same structure as `odepsmh`. The for loop becomes a while loop with the choice of step size `h` inserted after computing

```
coefs = fseries(t(k),y(:,k),deg)
```

and before updating `y(:,k+1)` using Horner's rule. We follow [12], which uses an adaptation of the root test for series, which gives radius of convergence $\rho = \lim_{j\to\infty}(1/a_j)^{1/j}$ for power series $\sum_{j=0}^{\infty} a_j(t - t_k)^j$. For each component `i` starting at `y(i,k)`, we have series $a_j =$ `coefs(i,j+1)`. The adaptation maximizes (infinity norm) and scales by s across components as follows, computing one $\rho_j = (s/a_j)^{1/j}$ for each j from 1 to `deg` (in our vector `roottests`).

```
scale = max(1,norm(y(:,k),inf));
roottests = (scale./vecnorm(coefs(:,2:end),inf,1)).^(1./(1:deg));
hm = min(roottests([deg-1,deg]))/exp(2) * exp(-0.7/(deg-1));
h = min(hm, min(roottests));
```

Since we can't compute the limit in the original root test and want a step that is significantly smaller than the radius of convergence, `hm` uses the minimum of the last two ρ_j terms and divides by e^2 as described above. The factor `exp(-0.7/(deg-1))` is a further step reduction for safety. Using the last two terms protects against a series where every other a_j could be very small (or zero), leading to an unreasonably large step. In the case that `hm` is still large compared to earlier ρ_j terms (e.g. a nearly quadratic solution), the final safety is to take at least the minimum of all computed ρ_j terms.

This is just one example of an adaptive scheme that uses generated series code. The above tests of `odepsmJZ` on the forced damped pendulum show good success in matching the requested tolerances of `1e-13` and `1e-3`, while the ode45 had no accuracy when using default relative tolerance of `1e-3` and absolute tolerance of `1e-6`.

7. Conclusion. Anyone studying or applying differential equations should be aware that a series recurrence relation algorithm is available for the solution of almost any ODE initial value problem. Finding the recurrence is methodical and a computer program can produce executable code for the recurrence around a variable initial point to variable order. In a differential equations class, students can see that the idea of a power series solution is not restricted to simple textbook examples of linear equations. Using the recurrence code, the radius of convergence can be estimated from the series coefficients and visualized by plotting different order Taylor polynomial solutions. When considering numerical methods, students or applied mathematicians can see the practicality of the Taylor series method to any desired higher order. Tests show that high-order methods can be competitive with other methods and MATLAB's ode45. The high-order methods are particularly useful in applications that require high accuracy [2].

Our MATLAB implementation (`https://github.com/rineidinger/psm4odes`) carries an open source license and may be used, studied, and modified as is helpful, with educational uses especially encouraged. A parallel exposition and development in Python could reach another audience, particularly in computer science. Other developers can develop (and have developed) professional tools that fine tune efficiency based on language and computing environment. There is much potential for further research into adaptive methods that can dynamically adjust the order used in addition to the step-size. One idea is using the *a priori* error bound; we've used a version of the trace class that also returns constants in the final bound in [21]. There seems to be boundless potential for the algorithmic Taylor method in solving ODEs.

REFERENCES

[1] A. Abad, R. Barrio, F. Blesa, M. Rodriguez, *Algorithm 924: TIDES, a Taylor Series Integrator for Differential EquationS*, ACM Trans. Math. Softw. 39 (2012), pp. 5:1-28. https://doi.org/10.1145/2382585.2382590

[2] D.H. Bailey, R. Barrio, J.M. Borwein, *High-precision computation: Mathematical physics and dynamics*, Applied Mathematics and Computation, 218 (2012), pp. 10106-10121. https://doi.org/10.1016/j.amc.2012.03.087.

[3] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, 10th ed., Wiley, Hoboken, NJ, 2012.

[4] R. L. Burden and J. D. Faires, *Numerical Analysis*, 8th ed., Thomson Brooks/Cole, Belmont, CA, 2005.

[5] Y. F. Chang and G. Corliss, *ATOMFT: Solving ODEs and DAEs Using Taylor Series*, Computers & Mathematics with Applications, 28 (1994), pp. 209-233.

[6] G. Corliss and Y. F. Chang, *Solving Ordinary Differential Equations Using Taylor Series*, ACM Trans. Math. Softw. 8 (1982), pp. 114–144, doi: 10.1145/355993.355995.

[7] S. A. Forth, *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB*, ACM Transactions on Mathematical Software, 32 (2006), pp. 195–222.

[8] A. Gibbons, *A Program for the Automatic Integration of Differential Equations using the Method of Taylor Series*, Computer J., 3 (1960), pp. 108-111. doi: 10.1093/comjnl/3.2.108.

[9] A. Griewank and G. F. Corliss, eds., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991.

[10] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., SIAM, Philadelphia, PA, 2008.

[11] J. Guenther and M. Wolf, sponsor P. Warne, *An Adaptive, Highly Accurate and Efficient, Parker-Sochacki Algorithm for Numerical Solutions to Initial Value Ordinary Differential Equation Systems*, SIAM SUIRO, 12 (2019), doi: 10.1137/19S019115.

[12] A. Jorba and M. Zou, *A Software Package for the Numerical Integration of ODEs by Means of High-Order Taylor Methods*, Experimental Math., 14 (2005), pp. 99-117, doi: 10.1080/10586458.2005.10128904.

[13] R.E. Moore, *Methods and Applications of Interval Analysis*, SIAM, 1979, Section 3.4.

[14] R. D. Neidinger, *Automatic Differentiation and APL*, College Math. J., 20 (1989), pp. 238-251.
[15] R.D. Neidinger, *Introduction to automatic differentiation and MATLAB object-oriented programming*, SIAM Rev. 52 (2010), pp. 545–563.
[16] R.D. Neidinger, *Efficient recurrence relations for univariate and multivariate Taylor series coefficients*, AIMS Conference Publications, 2013, 2013(special): 587-596. doi: 10.3934/proc.2013.2013.587.
[17] G.E. Parker & J.S. Sochacki, *Implementing the Picard Iteration*, Neural, Parallel and Scientific Computation 4 (1996), 97-112.
[18] L.B. Rall, *Automatic Differentiation: Techniques and Applications*, Lecture Notes in Computer Science 120, Springer-Verlag, Berlin, 1981.
[19] C. Simó, *Global Dynamics and Fast Indicators*, in Global Analysis of Dynamical Systems, H.W. Broer, B. Krauskopf, and G. Vegter, eds., IOP Pub., Bristol, 2001, pp. 373-389.
[20] J. Sochacki, A. Tongen, *Applying Power Series to Differential Equations*, Springer, Cham, Switzerland, 2022.
[21] R.J. Thelwell, P.G. Warne, D.A. Warne, *Cauchy-Kowalevski and polynomial ordinary differential equations*, Electron. J. Diff. Equ. (2012), pp. 11:1-8.
[22] J. Waldvogel, Der Tayloralgorithmus, *J. Applied Math. and Physics* (ZAMP) 35 (1984), 780-789.
[23] P.G.Warne, D.A. Polignone 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 system*s, Computers & Math, with Appl., 52 (2006), pp. 1695-1710, doi: 10.1016/j.camwa.2005.12.004.