Solving Ordinary Differential Equations with Recursive Taylor Series

Jim Armstrong
Singularity
October 2005

Abstract

I recently had need to numerically solve some first-order, ordinary differential equations (ODE's) as part of a prototyping effort for a Flash learning activity. Normally, this would be a routine coding effort, involving the use of a fourth-order Runga-Kutta method or perhaps an Adams-Moulton predictor-corrector. Instead, I decided to revive an old technique learned in my college days under Dr. R.E. Moore.

This paper provides an informal introduction to recursive Taylor series (RTS), a largely overlooked but often very powerful method for solving ODE's . An example code is provided as a separate download, written in Flash ActionScript.

1 Introduction

The fundamental problem of interest is the first-order, ordinary differential equation

$$x' = f(t,x) \qquad [1\text{-}1]$$

over an interval [a,b], where $x(a) = c$, is specified in advance. The function, x , is real-valued and the function, f , is analytic in [a,b].

This is commonly known as an initial-value problem. If the function, f, is relatively simple, a closed-form solution may be obtained for x using standard methods for solving ODE's. In many cases, analytic solutions are not possible and it is necessary to resort to numerical methods to approximate the solution.

Numerical methods for solving ODE's have a venerable history, dating back to the mid 1700s when Leonhard Euler published the technique that now bears his name. Although many algorithms have been published since that time, they are all essentially attempts to avoid the explicit evaluation of derivatives in the Taylor series expansion of f.

2 Taylor Series

Numerical solutions of initial-value problems begin at the known initial point and advance the solution in increments, h, requiring an approximate solution at some point $t+h$. The Taylor series expansion is a reasonable starting point since it provides the k-th order approximation

$$f(t+h) \approx f(t) + hf'(t) + \frac{h^2}{2!}f''(t) + \frac{h^3}{3!}f'''(t) + \ldots + \frac{h^k}{k!}f^k(t) \qquad [2\text{-}1]$$

Given a step-size, h, and choice of order, k, the solution can be approximated in increments of h, starting at the initial point, $t = a$.

The Taylor series approximation has an apparent drawback in that the k-th order method requires evaluation of up to the k-th order derivative a function. Derivatives can grow rapidly in complexity, even for relatively simple functions. The cost of explicit derivative evaluation in Taylor series has driven research into approximation methods. These methods have similar error properties without the requirement for derivative evaluation. Some of the popular methods include Runga-Kutta, Adams-Bashforth-Moulton, predicator-corrector schemes, Galerkin, and collocation methods.

Although many of these methods are quite powerful and very useful in production applications, the fundamental assumption on which all are based is not entirely accurate. It is not necessary to explicitly compute derivates in order to use the Taylor series approximation.

3 Recursive Taylor Series

Equation [2-1] can be rewritten as

$$f(t+h) \approx \sum_{i=0}^{k} (f)_i h^i \qquad (f)_i = \frac{f^{(i)}(t)}{i!} \qquad \text{[3-1]}$$

If the coefficients, $(f)_i$, are known, then the series approximation can be very efficiently computed via nested multiplication. Moore [1], popularized the use of recursion relations for generating the Taylor coefficients. Starting at $t = a$, the initial conditions provide $(f)_0$. Recursion relations provide formulas for the $(f)_i$ in terms of previous values so that the exact functional form of the derivatives is not required.

Equation [1-1] expresses information about $x(t)$ in terms of its derivative, $f(t,x)$. The numerical solution advances $x(t)$ to $x(t+h)$. This requires relating the Taylor coefficients of f to those of x, which is accomplished by the formula

$$(x)_j = (x')_{j-1} / j \qquad \text{[3-2]}$$

Equation [3-2] follows from the definition,

$$(x')_i = \frac{(x')^{(i)}}{i!} \qquad \text{[3-3]}$$

where $(x')^{(i)}$ is a shorthand notation for the i-th derivative of x', evaluated at t. From [3-3],

$$(x')_i = \frac{(x')^{(i)}}{i!}\frac{(i+1)}{(i+1)} = (i+1)\frac{x^{(i+1)}}{(i+1)!} = (i+1)(x)_{i+1} \qquad \text{[3-4]}$$

With a change of variable, $j = i+1$, [3-2] follows directly from [3-4]. Since $x' = f$, the Taylor coefficients of x are related to those of f by

$$(x)_j = j^{-1}(f)_{j-1} \qquad \text{[3-5]}$$

4 Elementary Recursion Relations

Recursion relations for complex equations can be constructed using combinations of relations for simpler functions. The following list of recursion relations is useful for a wide variety of functional relationships in [1-1]. *u(t)* and *v(t)* are analytic functions in the interval of interest.

$$(u \pm v)_k = (u)_k \pm (v)_k \qquad [4\text{-}1]$$

$$(uv)_k = \sum_{j=0}^{k} (u)_j (v)_{k-j} \qquad [4\text{-}2]$$

$$(u/v)_k = \frac{1}{v}\left((u)_k - \sum_{j=1}^{k} (v)_j (u/v)_{k-j} \right) \qquad [4\text{-}3]$$

$$(u^a)_k = \frac{1}{u} \sum_{j=0}^{k} (a - jk^{-1}(a+1))(u)_{k-j} (u^a)_j \qquad [4\text{-}4]$$

$$(e^u)_k = \sum_{j=0}^{k-1} (1 - jk^{-1})(e^u)_j (u)_{k-j} \qquad [4\text{-}5]$$

$$(\ln u)_k = \frac{1}{u}\left((u)_k - \sum_{j=1}^{k-1} (1 - jk^{-1})(u)_j (\ln u)_{k-j} \right) \qquad [4\text{-}6]$$

$$(\sin u)_k = k^{-1} \sum_{j=0}^{k-1} (j+1)(\cos u)_{k-1-j} (u)_{j+1} \qquad [4\text{-}7]$$

$$(\cos u)_k = -k^{-1} \sum_{j=0}^{k-1} (j+1)(\sin u)_{k-1-j} (u)_{j+1} \qquad [4\text{-}8]$$

The derivation of [4-2] provides some insight into derivation of the other formulas. From Leibnitz's rule for generalized products,

$$(uv)^{(k)} = \sum_{j=0}^{k} \left(\frac{k}{j} \right) u^{(j)} v^{(k-j)} = \sum_{j=0}^{k} \frac{k!}{j!(k-j)!} u^{(j)} v^{(k-j)} = k! \sum_{j=0}^{k} \frac{u^{(j)}}{j!} \frac{v^{(k-j)}}{(k-j)!}$$

From which

$$\frac{(uv)^{(k)}}{k!} = (uv)_k = \sum_{j=0}^{k} (u)_j (v)_{k-j}$$

The coupling in equations [4-7] and [4-8] results from coupling in the derivatives of the sine and cosine functions.

In practice, several of the recursion relations can be simplified. For example, common values of a in [4-4] are *a = -1, a =1/2* and *a = -1/2*. This yields,

$$(u^{-1})_k = -u^{-1} \sum_{j=0}^{k} (u^{-1})_j (u)_{k-j} \qquad \text{[4-4a]}$$

$$(\sqrt{u})_k = \frac{1}{2ku} \sum_{j=0}^{k} (k-3j)(\sqrt{u})_j (u)_{k-j} \qquad \text{[4-4b]}$$

$$(1/\sqrt{u})_j = \frac{1}{2ku} \sum_{j=0}^{k} (k-j)(1/\sqrt{u})_j (u)_{k-j} \qquad \text{[4-4c]}$$

In the case of $u = t$ in [4-7] and [4-8],

$$(\sin t)_k = k^{-1} (\cos t)_{k-1}$$

$$(\cos t)_k = -k^{-1} (\sin t)_{k-1}$$

The following example illustrates the practical use of the above equations.

Derive the recursion relations for the ODE, $x' = xe^t + \sin x$.

Solution:

1) Define the list of simple relations,

$$T_1 = e^t$$
$$T_2 = xT_1$$
$$T_3 = \sin x$$
$$T_4 = \cos x$$
$$x' = T_2 + T_3$$

Since e^t is its own derivative, $(T_1)_k = k^{-1}(T_1)_{k-1}$

From the basic recursion relations,

$$(T_2)_k = \sum_{j=0}^{k} (x)_j (T_1)_{k-j}$$
$$(T_3)_k = k^{-1} \sum_{j=0}^{k-1} (j+1)(T_4)_{k-1-j}(x)_{j+1}$$
$$(T_4)_k = -k^{-1} \sum_{j=0}^{k-1} (j+1)(T_3)_{k-1-j}(x)_{j+1}$$
$$(x')_k = (T_2)_k + (T_3)_k$$

Using [3-5], $(x)_k = k^{-1}\left[(T_2)_{k-1} + (T_3)_{k-1}\right]$

While explicit formulas for the derivatives are not required, RTS still requires the derivation of recursion relations. For many functions, this process can be automated using a combination of symbolic and numeric algorithms.

In many cases, there can be a computational savings from using RTS over a more traditional method. For example, consider a standard Runga-Kutta method of the form

$$x(t+h) = x(t) + \frac{h}{6}\left(A + 2B + 2C + D\right)$$
$$A = f(t, x)$$
$$B = f(t + h/2, x + h/2)$$
$$C = f(t + h/2, x + Bh/2)$$
$$D = f(t + h, x + hC)$$

Apply this approximation to the ODE, $x' = x^2 + t^2$. The R-K method requires 13 additions and 15 multiplications per step. The reader may verify that the explicit derivatives (up to fourth order) of the Taylor series expansion of x' do indeed grow rapidly in complexity. The operation count from the explicit derivative formulas is not competitive with R-K. RTS, however, when evaluated using nested multiplication, only requires 9 additions and 14 multiplications per step. The recursive nature of the computations also lends itself nicely to maintaining interim results in registers. By pre-computing and storing common operations across steps, the example program (available for download) illustrates how all results (coefficient computation and polynomial evaluation) can be computed ‘on the fly’ in compact loops.

This is not to say that RTS is always computationally superior to an equivalent-order R-K method or any other numerical method for solving ODE’s. This example, and other uses of RTS in practice, illustrate that the time required to manually derive recursion relations may yield a computation advantage in some problems. If the solution is run often with different starting points, or constant parameters that vary from run to run, then the derivation may also be worth the effort as the benefits are realized across multiple runs.

5 Numerical Example

This section derives recursion relations and illustrates an actual implementation of RTS, coded in Flash Actionscript. The sample problem is

$$x' = \sin t - 2x \quad , \; x(0) = 0 \qquad [5\text{-}1]$$

This is a non-homogeneous problem whose general solution can be obtained as the superposition of complementary and particular solutions. The complementary solution, x_c , is obtained by solving the homogeneous problem

$$x' + 2x = 0$$

yielding $x_c = e^{-2t} + c$

The particular solution, x_p, can be obtained with the method of undetermined coefficients. Take

$$A\sin t + B\cos t \qquad [5\text{-}2]$$

as a trial solution, from which

$$x_p' + 2x_p = A\cos t - B\sin t + 2(A\sin t + B\cos t) = (A + 2B)\cos t + (2A - B)\sin t$$

Since $x_p' + 2x_p = \sin t$,

$$2A - B = 1$$
$$A + 2B = 0$$

which yields the coefficients,

A = 2/5, B = -1/5

Substitute into [5-2], and then add complimentary and particular solutions to obtain the general solution,

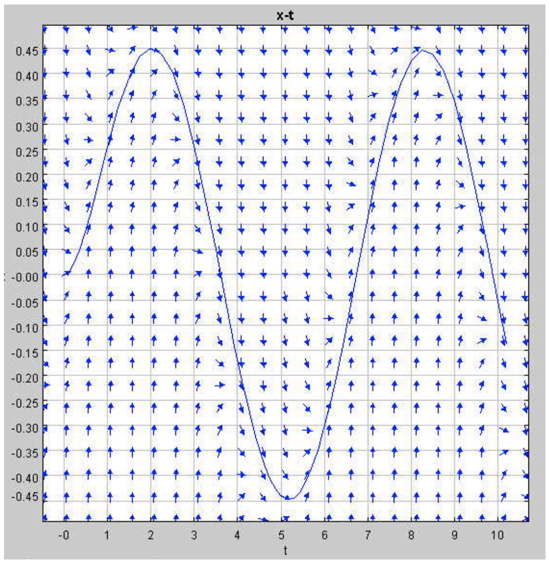
$$x(t) = \frac{1}{5}\left(2\sin t - \cos t\right) + e^{-2t} + C$$

C is chosen to fit the initial condition, $x(0) = 0$. It is easier, however, to use the form

$$x(t) = \frac{1}{5}\left(2\sin t - \cos t + e^{-2t}\right) \qquad [5\text{-}2]$$

which satisfies both the differential equation [5-1] and the initial condition.

As with many equations of this form, the homogeneous component goes rapidly to zero as t increases, so the general solution is dominated by the non-homogeneous component, which is a linear combination of sine and cosine. A graph of the solution and the direction field was obtained from ODE Toolkit (http://odetoolkit.hmc.edu).



To solve numerically via RTS, first derive the necessary recursion relations.

Define the list,

$$\begin{aligned} T_1 &= \sin t \\ T_2 &= \cos t \\ T_3 &= 2x \\ x' &= T_1 - T_3 \end{aligned}$$

The recursion relations are

$$\begin{aligned} (T_1)_k &= k^{-1}(T_2)_{k-1} \\ (T_2)_k &= -k^{-1}(T_1)_{k-1} \\ (T_3)_k &= 2(x)_k \\ (x)_k &= k^{-1}\left[(T_1)_{k-1} - (T_3)_{k-1}\right] \end{aligned}$$

Normally, one would compute the polynomial approximation in two steps – compute the Taylor coefficients, then evaluate the polynomial via nested multiplication. Nested multiplication requires all coefficients to be available in advance, so the computation may be initialized at the last coefficient. In order to take full advantage of the recursive nature of the Taylor coefficient generation, it is useful to compute everything 'on the fly.'

Since the h^j and k^{-1} values are recomputed every step, these values are pre-computed in arrays outside the main loop. This avoids redundant (and expensive) divides and allows the Taylor polynomial to be efficiently computed as soon as coefficients are available.

The following table illustrates results on the interval [0,1] with a fourth-order method and step size of 0.125.

t	Approximation	Actual
0	0.0	0.0
0.125	0.00719197591145833	0.00719051652250624
0.25	0.0264874764595034	0.0264852313022069
0.375	0.0548833822237808	0.0548807978001591
0.5	0.0898322285165974	0.0898295912978951
0.625	0.12914975968459	0.129147244647179
0.75	0.170946056723662	0.170943762264255
0.875	0.213574845064599	0.213572817951835
1.0	0.255596736329246	0.255594989396853

The following table provides results on the same interval with a step size of 0.0625.

t	Approximation	Actual
0	0.0	0.0
0.0625	0.00187365214029948	0.00187360551385134
0.125	0.00719059835150658	0.00719051652250624
0.1875	0.0155246194935666	0.0155245118428513
0.25	0.0264853571162695	0.0264852313022069
0.3125	0.0397142396943557	0.0397141019263159
0.375	0.0548809425287939	0.0548807978001591
0.4375	0.0716803178456389	0.0716801701320896
0.5	0.0898322285165974	0.0898295912978951
0.5625	0.10906680804995	0.109066663039864
0.625	0.129147385260078	0.129147244647179
0.6875	0.149843896799484	0.149843761936563
0.75	0.170943890420361	0.170943762264255
0.8125	0.192248805516489	0.192248684701023
0.875	0.21357293105451	0.213572817951835
0.9375	0.23474252688271	0.234742421654767
1.0	0.255595086755264	0.255594989396853

Note that for equivalent *t* in the first table, *x(t)* in the second table is not exactly the same, due to roundoff error. Although a smaller step size is theoretically more accurate (for a fixed order), accumulated roundoff error should be taken into account.

The following table illustrates results with the original step size of 0.125, but a fifth-order method. One of the benefits of RTS is that changing the order of the method does not alter the recursion relations, so varying the order of the approximation is trivial.

t	Approximation	Actual
0	0.0	0.0
0.125	0.00719045003255208	0.00719051652250624
0.25	0.0264851277858907	0.0264852313022069
0.375	0.0548806770120853	0.0548807978001591
0.5	0.0898322285165974	0.0898295912978951
0.625	0.129147123214447	0.129147244647179
0.75	0.170943649357666	0.170943762264255
0.875	0.213572716121794	0.213572817951835
1.0	0.255594899700782	0.255594989396853

6 Variable Order and Step Size

RTS is amenable to adaptive step size as with any other ODE method. Choice of optimal order and step size is discussed in [1] and [2] along with detailed error analysis. Moore showed that if $T_{k,h}$ is a Taylor-series approximation of order k and step-size, h and x^* is the exact solution, then

$$\left|T_{k,h} - x^*\right| \approx \left|T_{k,h} - T_{k,h/2}\right| \left(\frac{2^{k+1}}{2^{k+1}-1}\right) \qquad [6\text{-}1]$$

Interestingly, the data to evaluate [6-1] is available from the above tables and for a fourth-order method; the formula is accurate to single precision.

The considerations for RTS are essentially the same as any other method when attempting to solve stiff equations.

7 Summary

There is no such thing as a ‘one size fits all’ method for solving either initial-value or boundary-value problems for ODE’s. Although I have personally used RTS with great success for flight dynamics problems (stability and control), I have found it to be no better than a good implementation of R-K or a predictor-corrector method in other problems.

One of the early applications of RTS in [1] was numerical solution of the N-body problem to study the advance in the perihelion of Mercury as predicted by Special Relativity. The study compared measured data vs. a Newtonian simulation. This work formed the first computational validation of the theory of Special Relativity.

I hope that this informal discussion is useful for students taking their first course in numerical analysis as well as practitioners who are looking for alternatives to traditional methods. I very much enjoyed studying numerical analysis under Dr. Moore and also hope this paper is a small way of saying thanks for helping me learn to think outside the box.

References

[1] Moore, R.E., “Methods and Applications of Interval Analysis,” SIAM Studies in Applied Mathematics, SIAM, Philadelphia, 1979.

[2] Corliss, G.F. and Lowery, D., “Choosing a stepsize for the Taylor series method for solving ODE’s,” J. Computational Applied Math 3 (1977), pp. 251-256.

[3] Henrici, P., “Automatic computations with power series,” J. ACM 3 (1956), pp. 10-15.

[4] Kedem, G., “Automatic differentiation of computer programs,” ACM TOMS 6 (1980), pp. 150-163.