The Method of Modified Equations in Maple

Mohammad O. Ahmed and Robert M. Corless

Department of Applied Mathematics
University of Western Ontario, London, CANADA
{moa,rmc}@pineapple.apmaths.uwo.ca

Abstract

We discuss applying computer algebra to the analysis of numerical methods for initial value problems in ordinary differential equations (ODE). In particular, we show a Maple implementation of the method of modified equations, which is a technique for analyzing the effects of a specific numerical method applied to a specific problem. It has heretofore not been much used, because the calculations are tedious to do by hand, though reasonably straightforward with computer algebra. We show by several examples that this code helps us to learn useful things about the behaviour of numerical methods.

1 Numerical Methods for Initial Value Problems—A Review for Computer Algebraists

This paper discusses an application of computer algebra to the analysis of numerical methods for the solution of initial-value problems. In this section we lay out a brief review of developments in this area, because many computer algebraists may not be aware of the significant work that has been done in the last few decades. We do not give more than the highest of the highlights, and refer you to the excellent works [13, 14] for a concise but complete discussion.

This section can be skipped without loss of continuity.

1.1 Definitions

1. **Initial Value Problem (IVP)**: $\dot{y} = f(y)$, with $y(0) = y_0$, where $y(t) = [y_1(t), y_2(t), \ldots, y_N(t)]^T$ is N-dimensional. With no loss of generality we assume that the problem is autonomous (that is, f does not depend explicitly on t), by the usual trick of embedding the problem in one of dimension $N+1$ if necessary by putting the new variable $y_{N+1}(t) = t$ and adding the equation $\dot{y}_{N+1} = 1$ to f. Thus we may also assume without loss of generality that integration begins at $t = 0$. We will always assume that f is smooth enough, however, and this is a real restriction.

2. **Forward Error (also called the global error)**: $|y_{\text{computed}} - y_{\text{true}}|$, and all its variants. This error compares what is computed with the exact solution (which in general is unavailable and thus we must bound or estimate the forward error).

3. **Backward Error (also called the residual or defect or deviation)**: The amount by which the computed quantity fails to satisfy the original equation. That is, if the computed solution $y(t)$ exactly satisfies $y'(t) = f(y(t)) + r(t)$, then the residual $r(t) := y'(t) - f(y(t))$ is one way of looking at the backward error. In this paper we will concentrate on finding series expansions for the residual, for given numerical methods.

4. **Local (Truncation) Error**: This curious concept is one students and users have difficulty with. What users want (at least before they learn about residual and backward error) is the numerical method to give small (forward) 'global' error. Codes generally do not attempt to control the global error, and do not guarantee that it is small for the computed solution. One exception is the 'validated' (interval arithmetic) approach embodied in R. Lohner's code AWA [19].

   But the more usual floating-point codes do not provide answers with small global error guaranteed. Instead, they control the 'local error', which is the difference between the computed $y(t)$ on $t_k < t < t_{k+1}$ and the 'local exact solution' $u(t)$ which satisfies $\dot{u} = f(u)$, $u(t_k) = y_k$. Then one hopes that the control of the local error ensures that for a stable problem the global error is not too far distant from the estimated local errors. In practice this works reasonably well but the global error can sometimes be much, much larger than the local error.

   Our analysis of the method of modified equations starts with the conventional expression for the local truncation error.

5. **Differential-Algebraic Equation (DAE)** (not to be confused with an algebraic differential equation!): This is a mixture of differential equations and algebraic constraints. In some sense DAEs are infinitely 'stiff'. A simple example is the Rayleigh equation $\ddot{x} + \varepsilon(\dot{x} - 4\dot{x}^3/3) + x = 0$ written for $\varepsilon = \infty$ as

$$\begin{aligned} \dot{x} &= z \\ 0 &= z - \frac{4}{3}z^3 . \end{aligned} \tag{1}$$

To appear in the Electronic Proceedings of the 3rd International IMACS conference on Applications of Computer Algebra, Maui, July 24–26, 1997, Michael Wester and Stanly Steinberg, editors, `http://math.unm.edu/ACA/1997.html`. Send all communications to `rmc@pineapple.apmaths.uwo.ca`.

6. **Explicit Method**: a numerical method that does not need the solution of nonlinear equations to go from y_k to y_{k+1}. In this paper we use $y_{k+1} = y_k + h\psi(y_k, h)$. Examples include Euler forward and the classical 4th order Runge-Kutta method (RK4), as well as explicit Taylor series methods, extrapolation methods, and many exotic methods.

7. **Implicit Method**: a numerical method that requires the solution of nonlinear equations (depending on f) at each time step. Examples include Euler backward, and the 4th order Runge-Kutta method based on Gaussian points, below, which is an example of the infinite class of Implicit Runge-Kutta (IRK) methods.

$$\begin{aligned} Y_1 &= y_k + \frac{h}{4} f(Y_1) + h\left(\frac{1}{4} - \frac{\sqrt{3}}{6}\right) f(Y_2) \\ Y_2 &= y_k + h\left(\frac{1}{4} + \frac{\sqrt{3}}{6}\right) f(Y_1) + \frac{h}{4} f(Y_2) \\ y_{k+1} &= y_k + \frac{h}{2}\left(f(Y_1) + f(Y_2)\right) . \end{aligned} \tag{2}$$

8. **Stiffness**: "Stiffness is a confluence of a problem, a method, and a user's expectations."—A. Iserles. More prosaically, a stiff problem is one for which an implicit method gives answers with a small forward error for less computational effort than an explicit method can.

1.2 Main ideas behind current codes

1.2.1 Equidistribution

Probably the most important idea is that for efficiency, one should adjust the step size (or order or method or all three) so that the error made at each step is roughly the same ("equally distributed") all across the interval of integration. All modern codes for general purpose problems use this approach; for some special problems, it might be advantageous to use constant step sizes.

1.2.2 Efficient linear algebra

Many problems that occur in practice are stiff, and more economical solutions—often greatly more economical solutions—can be obtained by using implicit methods. But implicit methods require the solution of systems of nonlinear equations, and typically one would use a quasi-Newton method to do so. Thus at each time step, one would solve a sequence of linear systems. It turns out that efficiency in this process is crucial for obtaining overall efficiency. A very great deal of work has gone into the efficient solution of linear systems arising in the numerical solution of initial value problems, and indeed some methods are specially designed so that the linear algebra is easier (Singly-Diagonally Implicit Runge-Kutta or SDIRK methods, for example).

1.2.3 Interpolation

The classical way of solving differential equations numerically is to replace them with difference equations. Then iterating these difference equations gives us a sequence of points which are intended to be a discrete approximation to the solution of the differential equations. This works fine, for simple problems.

It turns out that for efficient graphical output (so that the solution curves look smooth instead of having 'corners'), interpolation is helpful. Also, if one is solving a system of differential equations with so-called 'g-stops', i.e. a condition like

$$\dot{y} = \begin{cases} f_1(y) & \text{if } g(y) > 0 \\ f_2(y) & \text{if } g(y) \le 0 \, , \end{cases} \tag{3}$$

then interpolation is required to find the place where $g(y) = 0$ and the differential equation thus changes character. One also needs interpolation for step-size changes or order changes or method changes in polyalgorithms. Finally, interpolation is useful for the solution of delay-differential equations.

Recently it has been realized (see the references to Wayne Enright's work in [8]) that an interpolant also returns us to the original concept of the solution: the computed solution is a continuous function, not a sequence of points. This means that we can substitute it back into the differential equation and compute the residual. We may now think about using this residual for error control, replacing the tricky idea of 'local error'.

1.2.4 Notions of error

Since the late 1980s it has become clearer that the *invariants* of a problem play a crucial role (see e.g. [11]), and numerical methods that preserve important invariants are often to be preferred over (even notionally otherwise more accurate) methods. Examples include Hamiltonian problems, which are often much better solved by symplectic methods [20].

Another difficulty arises for chaotic problems. A differential equation is *chaotic* if its solutions are bounded but initial perturbations grow exponentially (the Lyapunov exponents are positive). The classical theory of error in numerical methods suggested that numerical errors would also be amplified exponentially, and thus numerical solutions of chaotic problems ought to be worthless.

This is not so, and has been known to be not so since before the word 'chaos' became fashionable. Numerical simulations are remarkably faithful to quantities such as the (sometimes fractal) dimension of the attractor, or indeed the (Lebesgue-like) invariant measure on the attractor, or indeed to the Lyapunov exponents—all while getting trajectories that one could prove must be wrong. The difficulty was to explain the success of numerical methods in this case.

In many contexts, the explanation involves using a different measure of error from the forward error. The forward error grows exponentially, it is true, as the integration proceeds. However, a good numerical method gives you the exact solution of a nearby problem, and one can show that the backward error is (usually, for a variable-stepsize method with a reasonable step-size heuristic) uniformly small. Using 'defect control' as the error control, instead of the more usual local error control, ensures that this is so [8]. Going one step further, with more work one can often show that the computed solution is shadowed uniformly closely by a true solution—albeit one from a perturbed initial condition. Software now exists to do this *a posteriori* verification [5].

One difficulty with using the residual is that while nominally we have computed the exact solution to

$$\dot{y} = f(y) + \varepsilon v(t) \tag{4}$$

where ε is the user's tolerance and $||v|| < 1$, there are visible correlations of $v(t)$ with $y(t)$ (see [6]). That is, a more useful model of the residual is not $v(t)$ but $v(t, y)$. The method of modified equations will help us to predict and explain this correlation.

1.2.5 Software qualities

There have always been debates as to which is the 'best' numerical method for the solution of initial value problems. The main methods in use are multistep methods (implicit and explicit, variable step and variable order), Runge-Kutta methods (which have not yet reached their full potential), extrapolation methods, and the so-called General Linear Methods of Butcher and Burrage [3]. We would like to add Taylor series methods to that list, particularly because this fundamental approach is relevant to computer algebra packages (it is clearly the right way to compute arbitrary-precision answers, for example) but in spite of good existing codes (such as ATOMFT by Corliss and Chang [9], and the interval arithmetic code AWA by Lohner [19]), they are not popular.

There is a reason for this. It is very easy to write a quick and dirty (fixed time-step, fixed order) Runge-Kutta method, much easier than writing a Taylor series method (which has to use automatic differentiation efficiently to generate the Taylor series terms at each time-step, see [2] for an early description). But one gets free interpolants from the Taylor series methods, easy step-size selection, easy change of order; and implicit methods are exactly as hard (or as easy) as for Runge-Kutta methods. Once written, Taylor series codes are quite competitive with the main methods listed above.

If you add all these features (which you get almost for free with Taylor series) to your Runge-Kutta program, the difficulty in programming the method has risen quite remarkably, to the point where it is debatable which is harder. However, by the time you realize this you might be committed already to the Runge-Kutta (or multistep or extrapolation or GLM) approach.

We do not know of a clear-cut winner among the main methods. Some problems are best solved by multistep codes, some by Runge-Kutta methods, some by extrapolation methods. We do not know of a general-purpose implementation of General Linear Methods, but our knowledge of these methods is out of date and it is possible (indeed likely) that there are competitive implementations now in existence.

The performance of good codes (variable step size, variable order, variable method) do not vary much on the majority of problems, as far as we know.

Exotic special methods such as Kahan's [17] are occasionally of interest, and in such cases one might want to analyze the method for your problem by using the method of modified equations. We look at some of Kahan's methods in section 4.

1.2.6 Where to find good software

The first place to look is GAMS (Guide to Available Mathematical Software) at `http://gams.nist.gov`. Other places are the ftp site for the codes of [13, 14]. Codes that are particularly recommended are VODPK by George Byrne and others, LSODE/LSODI/DASSL by Hindmarsh and Petzold and others, and the codes of [13, 14]. For boundary value problems, it is very difficult to beat COLNEW by Ascher, Mattheij, and Russell. Finally, the Matlab ODE Suite by Shampine and Reichelt [21] is a very welcome addition to Matlab, replacing the old ODE45 and ODE23 with a suite of powerful methods specialized to take advantage of Matlab's interface and facility with vectors and matrices.

All of these codes represent a significant effort and many years of work by experts. The codes are robust and efficient, though—with the exception of the Matlab ODE suite—not as easy to use as a generation of programmers raised with proto-PSEs could wish.

1.3 Tools for analysis

The theory of error analysis for numerical methods of solution of differential equations is well-developed. Consult [13, 14] for details. The principal ideas are the notions of 'stability' or 'well-conditioning' of an initial-value problem, the use of Lipschitz constants and 'one-sided' Lipschitz constants, due to Dahlquist [10, 13], through which one can derive forward error bounds of the form

$$||y_T - y_C|| \leq e^{Lt} C h^p \tag{5}$$

where L is the Lipschitz constant, C is a constant, h is an average stepsize, and p is the order of the method. We see that on compact time intervals $0 \leq t \leq T$ the error converges to zero as $h \to 0$. This is the classical theory.

If $L > 0$ this theory is not helpful for long-time integration, and indeed $L \leq 0$ can only happen in the one-sided case.

Finally there is the Gröbner-Alexeev nonlinear variation of constants formula [13], which states that if the residual is $r(t)$, then there exists a smooth function G such that

$$y_T - y_C = \int_0^t G \cdot r(\tau) d\tau \tag{6}$$

which shows that G, in some sense, plays the role of a condition number for the problem.

2 The method of modified equations

> Series solutions have the advantage of being generally applicable, but are wholly devoid of elegance.
> —Stokes

In this section we explain how to compute a series expansion for the residual $r(t)$, given a specific numerical method for the solution of a particular ordinary differential equation. This is the method of modified equations, which goes back at least to Warming and Hyett in the 1970's, and more recently has been given an elegant and complete algebraic solution in terms of B-series by Hairer in 1993. We first give a hand example to explain the method, and then show the results of the Maple implementation. We use Taylor series, in a brute-force manner. We hope to implement the elegant approach of Hairer in the near future.

The method of modified equations is explained very well in [12], and is used in [4, 6] to explain long-term asymptotic behaviour and the growth of the global error for symplectic methods respectively. More examples are given in [1], along with a discussion and worksheet implementation for partial differential equations.

Modified equations are intended to *explain* the numerics, but not to excuse them.

2.1 A hand-worked example

We consider the fixed time-step iteration

$$y_{k+1} = y_k + h\lambda y_k \ , \tag{7}$$

We first replace y_k with $y(t)$ and y_{k+1} with $y(t+h)$, and following convention we divide by h. This gives us the so-called local truncation error.

$$\mathrm{LTE} := \frac{y(t+h) - y(t) - \lambda\, h\, y(t)}{h} \tag{8}$$

The key idea of the method of modified equations is that if we *set the local truncation error to zero*, this defines a (new) function $y(t)$ for which the numerical method is exact. This gives us a functional equation for $y(t)$, and we wish to find an equivalent differential equation, because we understand differential equations better than we do functional equations.

We begin by computing a series expansion of this functional equation. This gives

$$\sum_{\ell \geq 1} \frac{h^{\ell-1}}{\ell!} y^{(\ell)}(t) \ - \lambda y(t) = 0 \tag{9}$$

for this example. This is an infinite order modified equation for our problem, and of course it is a singular perturbation. To make progress, we truncate at some finite order.

This means that our modified equation will only be correct up to some order of the stepsize. In practice, this is usually enough to understand what a numerical method does to your problem on a compact time interval. Working to $O(h^3)$, we have

$$y'(t) = \lambda y(t) - \frac{h}{2} y'' - \frac{h^2}{6} y''' - \cdots \tag{10}$$

We now wish to find some way to eliminate the high-order derivatives on the right-hand side. We do this by repeated differentiation and elimination. One differentiation gives

$$y''(t) = \lambda y'(t) - \frac{h}{2} y''' - \cdots \tag{11}$$

and we take advantage of the fact that y'' appears in the first equation multiplied by h, and so we need only work to $O(h^2)$ in this equation. One more differentiation yields

$$y'''(t) = \lambda y''(t) - \cdots \tag{12}$$

and we may now begin to eliminate the higher derivatives, using these equations. We start with equation (12), and substitute it into equation (11) to get

$$y'' = \lambda y' - \frac{h}{2} \lambda y'' - \cdots \tag{13}$$

which we now *recursively substitute into itself* to get

$$y'' = \lambda y' - \frac{h}{2} \lambda^2 y' - \cdots . \tag{14}$$

We use equation (14) to simplify equation (12) to

$$y''' = \lambda^2 y' - \cdots . \tag{15}$$

At this point we have both y''' and y'' expressed in terms of lower derivatives.

We now substitute equations (15) and (14) into equation (10) to get

$$y' = \lambda y - \frac{h}{2}\left(\lambda y' - \frac{h\lambda^2}{2} y'\right) - \frac{h^2\lambda^2}{6} y' - \cdots \tag{16}$$

which we then substitute recursively into itself (twice) to get

$$y'(t) = \lambda y(t) - \frac{h\lambda^2}{2} y(t) + \frac{h^2\lambda^3}{3} y(t) - \cdots . \tag{17}$$

This is our $O(h^3)$ modified equation for the numerical method given at the start. We will discuss what it means in a later section.

2.2 Algorithm overview

The procedure just used is an algorithm. For the numerical method $y_{k+1} = y_k + h\psi(y_k, h)$ we first form the local truncation error (with $y(t)$ for y_k and $y(t+h)$ for y_{k+1})

$$\mathrm{LTE} = \sum_{\ell=1}^{N} \frac{h^{\ell-1}}{\ell!} y^{(\ell)}(t) - \psi(y(t), h) \tag{18}$$

(we should also expand ψ in a series in h, but for explicit methods ψ doesn't depend on h and so this is often trivial), rearrange to isolate y' and then differentiate N times, truncating one more series term each time, so that we have expressions for y', y'', and so on all the way up to $y^{(N)}$. The expression for $y^{(N)}$ will contain only derivatives of lower orders.

Now assume inductively that we have expressions for $y^{(i+1)}$, $y^{(i+2)}$, and so on up to $y^{(N)}$, in terms of $y^{(i)}$ and below.

We substitute all these expressions into our expression for $y^{(i)}$. This will now contain $y^{(i)}$ on the right hand side, but only multiplied by some power of h. We recursively substitute this expression into itself (using $O(\log N)$ steps as the powers double each time) until we have an expression for $y^{(i)}$ in terms of strictly lower derivatives only.

We then use this expression to eliminate $y^{(i)}$ from our expressions for the higher derivatives, and the inductive step is complete.

Our modified equation is the expression for $y' = y^{(1)}$ which results after this algorithm is executed.

This algorithm is implemented in the Maple program in Appendix A.

2.3 Cost estimates

Speaking crudely, the cost is dominated by the cost of computing the expressions for y'', y''', and so on up to $y^{(N)}$. Depending on how complicated ψ is, this can have an exponential (or worse) cost in N. Usually we will keep N small, unless ψ is particularly simple.

After the differentiations, there are $O(N^2 n)$ substitutions and simplifications, where n is the dimension of the vector y.

2.4 Discussion of the example

We now return to the example $y_{k+1} = y_k + h\lambda y_k$. We learned that an $O(h^3)$ modified equation for this is

$$y'(t) = \lambda(1 - \frac{z}{2} + \frac{z^3}{3} - \cdots) y(t) \tag{19}$$

where $z = h\lambda$. We see immediately that it is the size of $z = h\lambda$ that matters. As $h \to 0$ we see that this modified equation is asymptotic to $y' = \lambda y$, with error $O(h)$ for a fixed λ. This means that the equation is *consistent* with the differential equation $y' = \lambda y$. Since the residual is $O(h)$, we see that this is a first order method. The other main pillar of numerical analysis, *stability*, is not immediately available with this method (though we will see that some information can be obtained in this way).

But in its place, other things are more obvious. First, if λ is large, we see that we need to take h very small for accuracy. Moreover this nonuniformity is quantified. One cannot simply take $h = 0.01$, and then solve the equation several times with different λ. Finally, though the local error is $O(h^2)$, we see by the Gröbner-Alexeev nonlinear variation of constants formula that the global or forward error is $O(h)$, just like the residual, on compact time intervals.

We now hand over computation of modified equations to the Maple program in Appendix A. Hereafter we simply show results.

2.5 Maple examples

The program takes, as input, a method applied to a problem, the vector of unknowns, the dimension of the problem, the variable of integration, the step size parameter, and (optionally) the desired order of the modified equation.

For our first example, consider Euler's method applied to the (one-dimensional, nonlinear) problem $y' = y^2$, $y(0) = 1$. The current version of the code only expects vector input, so we must package this as a vector equation (with one dimension).

```
>  y := vector(1);
```

$$y := \mathrm{array}(1..1, [])$$

```
>  diffeq := diff(y[1](t),t) = y[1](t)^2;
```

$$\mathrm{diffeq} := \frac{\partial}{\partial t}\, y_1(t) = y_1(t)^2$$

```
>  Euler := vector(1,[y[1](t)^2]);
```

$$\mathrm{Euler} := \left[\, y_1(t)^2 \,\right]$$

Euler's method is $u_{k+1} = u_k + h\psi(u_k)$, where here $\psi(v) = v^2$. We pass the vector function to modeq.

```
>  answer := modeq( Euler, y, 1, t, h, 4);
```

$$\mathrm{answer} := \left[y_1(t)^2 - y_1(t)^3\, h + \frac{3}{2}\, y_1(t)^4\, h^2 - \frac{8}{3}\, y_1(t)^5\, h^3 + \frac{31}{6}\, y_1(t)^6\, h^4\right]$$

Notice that the example is nonlinear. The above calculation means that the iteration $u_{k+1} = u_k + hu_k^2$, which is what you get when you apply Euler's method to $y' = y^2$, is really a much better approximation to the *modified equation*

```
>  modified := diff(y[1](t),t) = answer[1];
```

$$\mathrm{modified} := \frac{\partial}{\partial t}\, y_1(t) = y_1(t)^2 - y_1(t)^3\, h + \frac{3}{2}\, y_1(t)^4\, h^2 - \frac{8}{3}\, y_1(t)^5\, h^3 + \frac{31}{6}\, y_1(t)^6\, h^4$$

In fact, the u_k will be $O(h^5)$ close to the solution of this equation, but only $O(h)$ close to the solution of the original equation.

For a more complete discussion of this equation, see [6]. In brief, after we computed a few more terms in this equation, Bruno Salvy recognized the sequence of rational numbers appearing in the modified equation $y' = B(hy)y^2$. He recognized the series for two reasons: one, he had a prepublication version of [22], and two, Gilbert Labelle, who is the author of [18] which is the reference in [22] for this sequence, was visiting Bruno Salvy at the time. Labelle had solved this problem in some generality, in formal power series: modified equations for (scalar) numerical methods that can be nondimensionalized so that the second nonzero term of the series for $y + h\psi(y, h)$ is ± 1 (here $y_{k+1} = y_k + hy_k^2$ becomes $v_{k+1} = v_k + v_k^2$ on multiplication by h and defining $v = hy$) can be solved completely in formal power series. We highly recommend the paper [18].

From this formal power series solution, which gives all the coefficients in an asymptotic series for $B(hy)$ by recurrence, and by a convergent infinite product, one can show that the modified equation has a natural boundary of singularities in the complex plane, on a fractal.

We now return to the details of verifying that the finite-order modified equation we have in hand actually models the numerical method. We solve the equation with ten steps of Euler's method, and compare with the exact solution and with a good solution of the modified equation.

```
> u := array(0..10):
> u[0] := 1:
> for i to 10 do
>   u[i] := u[i-1] + 0.1*u[i-1]^2;
> od:
> pts := [seq([i/10.,u[i]], i=0..10)]:
> dataplot := plot(pts, style=POINT,
                        symbol=CIRCLE):
```

```
>  dsolve({diffeq,y[1](0)=1},y[1](t));
```

$$y_1(t) = -\frac{1}{t-1}$$

```
>  exactplot := plot(1/(1-t),t=0..1):
```

```
> modsol := dsolve({subs(h=0.1, modified),
              y[1](0)=1}, y[1](t), numeric):
```

```
>  with(plots):
```

```
> modplot := odeplot(modsol, [t,y[1](t)], 0..1):
> display({dataplot,exactplot}, view=[0..1,0..5]);
> display({dataplot, exactplot, modplot},
          view=[0..1,0..5]);
```

The results of these display commands appear in Figures 1 and 2. Notice the much better agreement between the numerical solution and the solution of the modified equation. That shows that the modified equation explains what Euler's method did to the original problem.

Now we try a more complicated problem. We consider the differential equation (which can be solved exactly in terms of Airy functions, see [15])

```
>  diff(y[1](t),t) = y[1](t)^2 - t;
```

$$\frac{\partial}{\partial t}\, y_1(t) = y_1(t)^2 - t$$

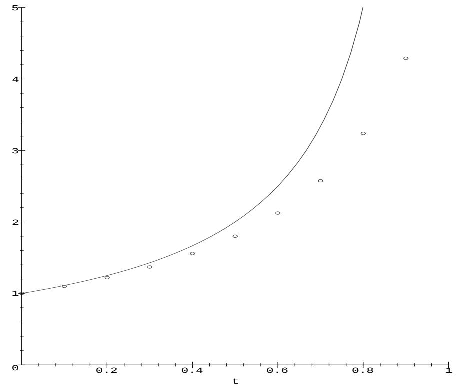

Figure 1: Exact solution $y = 1/(1-t)$ (—) and numerical solution by Euler's method with $h = 0.1$ (∘) to $y' = y^2, y(0) = 1$.

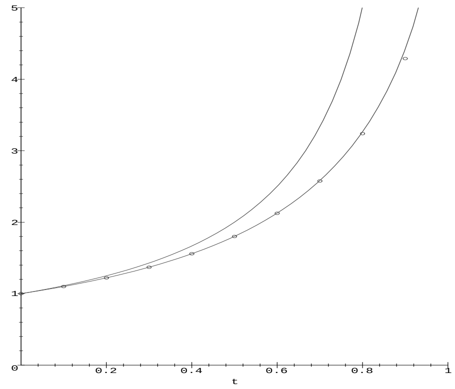

Figure 2: Exact solution and numerical solution by Euler's method with $h = 0.1$ as in Figure 1, and the solution of the modified equation $y' = (1 - hy + 3h^2y^2/2 - 8h^3y^3/3 + 31h^4y^4/6)y^2$, $y(0) = 1$, with $h = 0.1$. Notice that the solution of the modified equation nearly interpolates the Euler's method solution of the original equation.

and the following 2nd order Runge-Kutta method.

```
> k1 := y[1](t)^2 - t;
```

$$\mathrm{k1} := y_1(t)^2 - t$$

```
> k2 := (y[1](t) + h/2*k1)^2 - (t+h/2);
```

$$\mathrm{k2} := (y_1(t) + \frac{1}{2}\,h\,(y_1(t)^2 - t))^2 - t - \frac{1}{2}\,h$$

```
> mrk2 := modeq(vector(1,[k2]), y, 1, t, h, 4):
> series(mrk2[1],h,4):
> map(factor,");
```

$$(y_1(t)^2 - t) + (-\frac{3}{4}\,y_1(t)^4 + \frac{5}{6}\,y_1(t)^2\,t - \frac{1}{12}\,t^2 + \frac{1}{3}\,y_1(t))\,h^2 + \frac{1}{4}\,y_1(t)\,(5\,y_1(t)^4 - 6\,y_1(t)^2\,t + t^2 - 2\,y_1(t))\,h^3 + O(h^4)$$

We see several things immediately. First, the modified equation is $O(h^2)$ different from the original equation, and thus the numerical method is of second order accuracy. Second, the modification contains terms depending on the independent variable t, and thus the accuracy of the method cannot be uniform in t. Indeed we see that we must have $t < O(1/h)$ to have any hope of accuracy.

Now let us try a simple two-dimensional problem, the simple harmonic oscillator, solved by Euler's method, RK2, and by the classical RK4 method. We encode the differential equation $y'' + y = 0$ as usual by the first order system

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}. \tag{20}$$

```
> y := vector(2):
> shosc := vector(2, [y[2](t),-y[1](t)]);
```

$$\mathrm{shosc} := [\, y_2(t),\ -y_1(t) \,]$$

```
> meuler := modeq(shosc, y, 2, t, h, 3);
```

$$\mathrm{meuler} := \left[y_2(t) + \frac{1}{2}\,y_1(t)\,h - \frac{1}{3}\,y_2(t)\,h^2 - \frac{1}{4}\,y_1(t)\,h^3,\right. \left. -y_1(t) + \frac{1}{2}\,y_2(t)\,h + \frac{1}{3}\,y_1(t)\,h^2 - \frac{1}{4}\,y_2(t)\,h^3 \right]$$

```
> M := genmatrix([meuler[1],meuler[2]],
> [y[1](t),y[2](t)]);
```

$$M := \begin{bmatrix} \frac{1}{2}\,h - \frac{1}{4}\,h^3 & 1 - \frac{1}{3}\,h^2 \\ -1 + \frac{1}{3}\,h^2 & \frac{1}{2}\,h - \frac{1}{4}\,h^3 \end{bmatrix}$$

In order to understand this matrix physically, we look at the general equation $y' = Ay$, $y(0) = y_0$, which has the solution $y = y_0 \exp(At)$. So let us take the exponential of a matrix which has the same symmetry properties as the above.

```
> A := matrix(2,2,[b, a, -a, b]);
```

$$A := \begin{bmatrix} b & a \\ -a & b \end{bmatrix}$$

```
> exponential(A,t);
```

$$\begin{bmatrix} e^{(t\,b)}\cos(t\,a) & e^{(t\,b)}\sin(t\,a) \\ -e^{(t\,b)}\sin(t\,a) & e^{(t\,b)}\cos(t\,a) \end{bmatrix}$$

This shows that the diagonal term is a rate of damping (if $b < 0$) or exponential growth (if $b > 0$). Comparison with our modified equation shows that Euler's method has introduced exponential growth into the solution. In many cases, one will be able to identify such behaviour in the modified equation by comparison with known results.

Now we try RK2.

```
> f := v -> vector(2, [v[2],-v[1]]);
```

$$f := v \to \mathrm{vector}(2, [v_2, -v_1])$$

```
> k1 := f([y[1](t),y[2](t)]);
```

$$\mathrm{k1} := [\, y_2(t),\ -y_1(t)\,]$$

```
> k2 := f(evalm( vector(2,[y[1](t),y[2](t)]) +
h/2*k1 ));
```

$$\mathrm{k2} := \left[y_2(t) - \frac{1}{2}\, y_1(t)\, h,\ -y_1(t) - \frac{1}{2}\, y_2(t)\, h \right]$$

```
> modRK2 := modeq(k2, y, 2, t, h, 4);
```

$$\mathrm{modRK2} := \left[y_2(t) + \frac{1}{6}\, y_2(t)\, h^2 + \frac{1}{8}\, y_1(t)\, h^3 - \frac{1}{20}\, y_2(t)\, h^4, \right.$$
$$\left. -y_1(t) - \frac{1}{6}\, y_1(t)\, h^2 + \frac{1}{8}\, y_2(t)\, h^3 + \frac{1}{20}\, y_1(t)\, h^4 \right]$$

```
> MRK2 := genmatrix([modRK2[1],modRK2[2]],
> [y[1](t),y[2](t)]);
```

$$\mathrm{MRK2} := \begin{bmatrix} \frac{1}{8}\, h^3 & 1 + \frac{1}{6}\, h^2 - \frac{1}{20}\, h^4 \\ -1 - \frac{1}{6}\, h^2 + \frac{1}{20}\, h^4 & \frac{1}{8}\, h^3 \end{bmatrix}$$

We see that this method also introduces exponential growth, but on an $O(h^{-3})$ time scale, not an $O(1/h)$ time scale. Now we try RK4.

```
> k3 := f( evalm( vector(2,[y[1](t),y[2](t)])
+ h/2*k2));
```

$$\mathrm{k3} := \left[y_2(t) + \frac{1}{2}\, h\, (-y_1(t) - \frac{1}{2}\, y_2(t)\, h), \right.$$
$$\left. -y_1(t) - \frac{1}{2}\, h\, (y_2(t) - \frac{1}{2}\, y_1(t)\, h) \right]$$

```
> k4 := f( evalm( vector(2,[y[1](t),y[2](t)])
+ h*k3));
```

$$\mathrm{k4} := \left[y_2(t) + h\, (-y_1(t) - \frac{1}{2}\, h\, (y_2(t) - \frac{1}{2}\, y_1(t)\, h)), \right.$$
$$\left. -y_1(t) - h\, (y_2(t) + \frac{1}{2}\, h\, (-y_1(t) - \frac{1}{2}\, y_2(t)\, h)) \right]$$

```
> RK4 := evalm( k1/6 + k2/3 + k3/3 + k4/6):
> modRK4 := modeq(RK4, y, 2, t, h, 6):
> genmatrix([modRK4[1],modRK4[2]],
> [y[1](t),y[2](t)]);
```

$$\begin{bmatrix} -\frac{1}{144}\, h^5 & 1 - \frac{1}{120}\, h^4 + \frac{1}{336}\, h^6 \\ -1 + \frac{1}{120}\, h^4 - \frac{1}{336}\, h^6 & -\frac{1}{144}\, h^5 \end{bmatrix}$$

We see an intriguing difference. Classical RK4 introduces positive damping, not exponential growth; and again the time scale is much longer. Another qualitative difference from Euler's method is that the amplitude errors are actually $O(h^5)$ and the phase errors are $O(h^4)$ here, whereas for Euler's method the amplitude errors were $O(h)$ and the phase errors were $O(h^2)$.

For the final example in this section we consider the Van Der Pol equation.

```
> vdp := diff(v(t),t,t) -
epsilon*(1-v(t)^2)*diff(v(t),t) + v(t);
```

$$\mathrm{vdp} := (\frac{\partial^2}{\partial t^2}\, \mathrm{v}(t)) - \varepsilon\, (1 - \mathrm{v}(t)^2)\, (\frac{\partial}{\partial t}\, \mathrm{v}(t)) + \mathrm{v}(t)$$

We convert `vdp` to first order form in the usual way.

```
> f := v -> vector(2, [v[2],
epsilon*(1-v[1])*v[2] - v[1]]);
```

$$f := v \to \mathrm{vector}(2, [v_2, \varepsilon\, (1 - v_1)\, v_2 - v_1])$$

```
> v := vector(2);
```

$$v := \mathrm{array}(1..2, [])$$

```
> vt := vector(2, [v[1](t), v[2](t)]):
> k1 := f(vt);
```

$$\mathrm{k1} := [\, v_2(t),\ \varepsilon\, (1 - v_1(t))\, v_2(t) - v_1(t)\,]$$

```
> k2 := f(evalm(vt + h/2*k1));
```

$$\mathrm{k2} := \left[v_2(t) + \frac{1}{2}\, h\, (\varepsilon\, (1 - v_1(t))\, v_2(t) - v_1(t)), \varepsilon \right.$$
$$(1 - v_1(t) - \frac{1}{2}\, h\, v_2(t))$$
$$(v_2(t) + \frac{1}{2}\, h\, (\varepsilon\, (1 - v_1(t))\, v_2(t) - v_1(t))) - v_1(t)$$
$$\left. - \frac{1}{2}\, h\, v_2(t) \right]$$

```
> modvdpEuler := modeq(k1, v, 2, t, h, 1);
```

$$\mathrm{modvdpEuler} := \left[v_2(t) + (-\frac{1}{2}\, \%1 + \frac{1}{2}\, v_1(t))\, h, \%1 \right.$$
$$- v_1(t) + (\frac{1}{2}\, \varepsilon\, v_2(t)^2 + \frac{1}{2}\, \varepsilon\, (\%1 - v_1(t))\, v_1(t)$$
$$\left. - \frac{1}{2}\, \varepsilon\, (\%1 - v_1(t)) + \frac{1}{2}\, v_2(t))h \right]$$
$$\%1 := \varepsilon\, (1 - v_1(t))\, v_2(t)$$

As with the Airy function example we see nonuniformity, though this time with respect to ε. If ε is large, then h must be small for this modified equation to be close to the true solution.

```
> modvdpRK2 := modeq(k2, v, 2, t, h, 2):
> map(collect,",h,factor);
```

$$\Big[(-\frac{1}{6}\varepsilon^2 v_2(t) + \frac{1}{3}\varepsilon^2 v_2(t)\, v_1(t) + \frac{1}{6}\varepsilon\, v_1(t) - \frac{1}{6}\varepsilon^2 v_2(t)\, v_1(t)^2 - \frac{1}{6}\varepsilon\, v_1(t)^2 + \frac{1}{6}\varepsilon\, v_2(t)^2 + \frac{1}{6} v_2(t))h^2 + v_2(t), (\frac{1}{3}\varepsilon\, v_2(t) - \frac{7}{12}\varepsilon\, v_2(t)\, v_1(t) + \frac{1}{6}\varepsilon^3 v_1(t)^3 v_2(t) - \frac{1}{6}\varepsilon^3 v_2(t) + \frac{1}{6}\varepsilon^2 v_1(t) - \frac{1}{3}\varepsilon^2 v_1(t)^2 + \frac{1}{6}\varepsilon^2 v_1(t)^3 + \frac{1}{2}\varepsilon^3 v_2(t)\, v_1(t) - \frac{1}{2}\varepsilon^3 v_2(t)\, v_1(t)^2 - \frac{5}{12}\varepsilon^2 v_2(t)^2 v_1(t) + \frac{5}{12}\varepsilon^2 v_2(t)^2 - \frac{1}{6} v_1(t))h^2 + \varepsilon\, v_2(t) - \varepsilon\, v_2(t)\, v_1(t) - v_1(t)\Big]$$

Now we see some even *worse* nonuniformity, with powers of ε^3 coming in. This means that for large ε, this second order method may actually produce worse errors and require smaller time steps than Euler's method.

3 Implicit methods

Consider the following general method for doing modified equations for implicit methods. The idea is that we solve the implicit equations in series first, before we call `modeq`. This always works. Solving the equation $y_{k+1} = y_k + h\psi(y_k, y_{k+1})$ in series is most natural by an iterative method, either by simple fixed point iteration $y_{k+1}^{(m+1)} = y_k + h\psi(y_k, y_{k+1}^{(m)})$ or Newton's method, with an initial guess $y_{k+1}^{(0)} = y_k$.

For our first example take the equation $y'' + \sin(y) = 0$, solved by the implicit midpoint rule $v_{k+1} = v_k + hf((v_k + v_{k+1})/2)$. (An alternate formulation is $V = v_k + hf(V)/2$ followed by $v_{k+1} = v_k + hf(V)$). [This method is the only method which is symplectic, introduces no spurious fixed points, no spurious Hopf bifurcations, and no spurious flip bifurcations [16].]

The Maple session to find a modified equation is as follows.

```
> v := vector(2):

 > iter := proc(V::vector)
     local w; global v;
     w := vector(2);
     w[1] := convert(series(v[1](t) +
        h*(v[2](t)+V[2])/2, h),polynom);
     w[2] := convert(series(v[2](t) - h*sin(
        (v[1](t)+V[1])/2), h), polynom);
     eval(w)
   end:

> vt := vector(2, [v[1](t),v[2](t)]);
```

$$vt := [\, v_1(t),\, v_2(t)\,]$$

```
> i1 := iter(vt);
```

$$i1 := [\, v_1(t) + h\, v_2(t),\, v_2(t) - h\sin(v_1(t))\,]$$

```
> i2 := iter(i1);
```

$$i2 := \Big[v_1(t) + h\, v_2(t) - \frac{1}{2}\sin(v_1(t))\, h^2, v_2(t) - h\sin(v_1(t)) - \frac{1}{2}\cos(v_1(t))\, v_2(t)\, h^2 + \frac{1}{8}\sin(v_1(t))\, v_2(t)^2 h^3 + \frac{1}{48}\cos(v_1(t))\, v_2(t)^3 h^4 - \frac{1}{384}\sin(v_1(t))\, v_2(t)^4 h^5\Big]$$

```
> i3 := iter(i2):
> i4 := iter(i3):
> i5 := iter(i4):
> i6 := iter(i5):
> evalm(i6-i5);
```

$$[0,\, 0]$$

```
> psi := evalm((i6-vt)/h):
> modeq(psi,v, 2, t, h, 4):
> map(m->collect(m,h,factor),");
```

$$\Big[-\frac{1}{144} v_2(t) (6\cos(v_1(t))^2 - 6\sin(v_1(t))^2 + \cos(v_1(t))\, v_2(t)^2)\, h^4 - \frac{1}{12}\cos(v_1(t))\, v_2(t)\, h^2 + v_2(t), \frac{1}{144}\sin(v_1(t)) (6\cos(v_1(t))\, v_2(t)^2 + 6\cos(v_1(t))^2 - v_2(t)^4)\, h^4 + \frac{1}{24}\sin(v_1(t))\, (-v_2(t)^2 + 2\cos(v_1(t)))\, h^2 - \sin(v_1(t))\Big]$$

```
> map(m->collect(m,h,factor),i4);
```

$$\Big[\frac{1}{96} v_2(t)\, (\cos(v_1(t))\, v_2(t)^2 - 6\sin(v_1(t))^2)\, h^5 + \frac{1}{16}\sin(v_1(t))\, (v_2(t)^2 + 2\cos(v_1(t)))\, h^4 - \frac{1}{4}\cos(v_1(t))\, v_2(t)\, h^3 - \frac{1}{2}\sin(v_1(t))\, h^2 + h\, v_2(t) + v_1(t), -\frac{1}{384}\sin(v_1(t)) (-12\sin(v_1(t))^2 + 48\cos(v_1(t))\, v_2(t)^2 + v_2(t)^4)\, h^5 + \frac{1}{48} v_2(t) (6\cos(v_1(t))^2 - 6\sin(v_1(t))^2 + \cos(v_1(t))\, v_2(t)^2)\, h^4 + \frac{1}{8}\sin(v_1(t))\, (v_2(t)^2 + 2\cos(v_1(t)))\, h^3 - \frac{1}{2}\cos(v_1(t))\, v_2(t)\, h^2 - h\sin(v_1(t)) + v_2(t)\Big]$$

We see that the implicit midpoint rule is second order, and has a purely even power series expansion for the residual, has nonuniformities in the error (depending on the size of v), and with a little more work we can show that the Hamiltonian for the modified equation is of the form $H = H_0 + h^2 H_2 + h^4 H_4 + \cdots$.

As another example, we consider the nonlinear problem $y' = y^2$, $y(0) = 1$, solved by the 4th order two-stage Gauss method given in equations (2) in section 1. Solving equation (2) in series for y_{k+1} by iteration gives (with y_k replaced by $y(t)$, and abbreviated as y, and y_{k+1} replaced by $y(t+h)$, as usual)

$$y(t+h) = y + h\Big(y^2 + hy^3 + h^2y^4 + h^3y^5 + h^4y^6 + h^5y^7 + \frac{433}{432}h^6y^8 + \frac{437}{432}h^7y^9 + \cdots\Big).$$

If we now compute the modified equation to $O(h^8)$, we get

$$y'(t) = y^2(t) + \frac{h^6}{432}y^8(t) + O(h^8)\,. \tag{21}$$

Surprisingly, the normally 4th order Gauss method is actually 6th order accurate for this equation. However, it is clear that if $hy = O(1)$ then something will go wrong. Thus if y gets large, this analysis shows that the modified equation will not be close to the original one.

We wrote a short Matlab procedure to implement this method for this problem. Inversion of the Runge-Kutta matrix[1] gives the following system of equations:

$$\begin{aligned} Y_1^2 &= \left(3Y_1 + (2\sqrt{3}-3)Y_2 - 2\sqrt{3}y_k\right)/h \\ Y_2^2 &= \left(-(2\sqrt{3}+3)Y_1 + 3Y_2 + 2\sqrt{3}y_k\right)/h\,. \end{aligned} \tag{22}$$

This is already a total-degree order Gröbner basis for the nonlinear system of equations, and thus solution is easy (indeed, here one can even write the roots explicitly by means of radicals, though this is unstable numerically). There are four pairs of roots (Y_1, Y_2) for this system, only one pair of which approaches (y_k, y_k) as $h \to 0$. One can show that if $hy_k \approx 0.75$, two of these roots coalesce; in that case, the relationship of the roots of these nonlinear equations to the original differential equation becomes lost. This coalescence is an effect of nonlinearity, and it confirms (indeed underlies) the prediction of the method of modified equations that something breaks down when $hy = O(1)$.

4 An exotic method due to Kahan

In [17], we find the following methods: $y_{k+1} = Q(h, t_k, y_k)$, where

$$Q(\theta,\tau,g) := g + \frac{\tau + \theta/2 + g^2}{1/\theta - g} \tag{23}$$

[1] This is not always possible, because only some Runge-Kutta matrices are invertible, but when it is possible (as it is for Gauss methods) then there are computational and theoretical advantages in doing so. One possible advantage is that for solution of polynomial systems $\dot{y} = f(y)$ by implicit methods with invertible RK matrices, the inverted form of the equations will often already be a total-degree Gröbner basis, which conceivably could be useful numerically.

and

$$Q(\theta,\tau,g) := g + \frac{\tau + \theta/2 + g^2}{\frac{\sqrt{\tau+\theta/2}}{\tan\left(\theta\sqrt{\tau+\theta/2}\right)} - g} \tag{24}$$

It is straightforward to show using `modeq` that these methods solve

$$y' = t + y^2 + \frac{1}{6}yh^2 - \frac{1}{4}\left(y^2 + \frac{t}{3}\right)h^3 + \cdots \tag{25}$$

and

$$y' = t + y^2 + \left(\frac{y}{6} - \frac{ty^2}{3} - \frac{t^2}{3}\right)h^2 + \cdots \tag{26}$$

respectively. However, the special properties of these methods are not immediately apparent from this analysis, though we can confirm that they are both second order methods for solving $y' = t + y^2$.

5 Variable step size

The method of modified equations also applies to variable step size problems, as shown in [12]. The basic idea is that the error control heuristics used by the codes can be modelled by equidistribution of a monitor function, say $M(t, y(t))$ or $M(t, y(t), y'(t))$, and we can think of this as taking equal time steps in another variable, namely

$$\tau = \frac{\int_0^t M(t', y(t'))\,dt'}{\int_0^T M(t', y(t'))\,dt'} \tag{27}$$

We don't pursue the details here, but we remark that a good understanding of the heuristics used in the code is necessary to make this analysis work, and hence we expect this to be more useful for designers of codes than users.

In the case of *defect control*, however, the method of modified equations gives a series approximation for the defect and this turns the previous analysis on its head: we can use the method of modified equations to see what is being equidistributed. For the modified equation (21) above, a defect controlled code would make h^6y^6 approximately constant (approximately 432 times the tolerance, in fact) over each time step. This type of analysis might allow construction of good initial meshes for multigrid methods, for example.

6 Concluding remarks

The method of modified equations constructs a differential equation from a numerical method, from which the user may directly infer results about the behaviour of numerical method. It is important to note that while the modified equation is always more complicated than the original equation, it never needs to actually be solved—the idea is to use it directly, by looking at the new terms introduced into the equation and interpreting them 'physically'.

The method of modified equations has a long history, but is not much known to users, probably because of the tedium of computing modified equations by hand. We have shown that use of computer algebra to alleviate that tedium can allow greater advantage to be taken of the user's physical knowledge of the particular problem being modelled.

Acknowledgements

The authors are grateful to J. M. Sanz-Serna for discussions some years ago about the method of modified equations, to Greg Reid for comments on an earlier draft of this paper, and to V. Gerdt and F. Schwarz for the invitation to present this paper at the IMACS-ACA meeting in Maui, and to Michael Wester and Stanly Steinberg for organizing the meeting. We are also grateful to our colleague Henning Rasmussen for continuing discussions and encouragement on this and many other topics.

References

[1] Mohammad O. Ahmed, *An Exploration of Compact Finite Difference Methods For The Numerical Solution of PDE*, Ph.D. Thesis, Applied Mathematics, The University of Western Ontario, 1997.

[2] David Barton, I. M. Willers, and R. V. M. Zahar, "The automatic solution of ordinary differential equations by the method of Taylor series", Computer J. **14** vol. 3 *pp.* 243–248, 1970.

[3] John Butcher, *The Numerical Solution of Initial-Value Problems: Runge-Kutta and General Linear Methods*, Wiley, Chichester, 1987.

[4] M. P. Calvo, A. Murua, and J. M. Sanz-Serna, "Modified equations for ODEs" , Proceedings of Chaotic Numerics, Geelong, 1993, P. Kloeden and K. Palmer, eds, AMS Contemporary Mathematics **172** *pp.* 63–74, 1994.

[5] S.-N. Chow and Erik Van Vleck, "A shadowing lemma approach to global error analysis for initial value ODEs", SIAM J. Sci. Stat. Comp., **15** *pp.* 959–976, 1994.

[6] Robert M. Corless, "Error Backward", Proceedings of Chaotic Numerics, Geelong, 1993, P. Kloeden and K. Palmer, eds, AMS Contemporary Mathematics **172** *pp.* 31–62, 1994.

[7] Robert M. Corless, "What good are numerical solutions of chaotic differential equations", Computers in Mathematics with Applications, **28** 10–12, *pp.* 107–121, 1994.

[8] Robert M. Corless, "Defect-Controlled Numerical Methods and Shadowing for Chaotic Differential Equations", Physica D **60** *pp.* 323–334, 1992.

[9] George F. Corliss and Y. F. Chang, "Solving ordinary differential equations using Taylor series", ACM Trans. Math. Software, **8** vol. 2 *pp.* 114–144, 1982.

[10] Germund Dahlquist, *Stability and Error Bounds in the Numerical Integration of Ordinary Differential Equations*, Trans. Royal Inst. Tech., Stockholm, Sweden, No. 130, 1959.

[11] C. W. Gear, "Invariants and numerical methods for ODEs", Physica D **60**, *pp.* 303–310, 1992.

[12] D. F. Griffiths and J. M. Sanz-Serna, "On the scope of the method of modified equations", SIAM J. Sci. Stat. Comput. **7** *pp.* 994–1008, 1986.

[13] Ernst Hairer, Syvert P. Norsett, and G. Wanner, *The Numerical Solution of Ordinary Differential Equations*, volume 1 (Nonstiff problems), Springer-Verlag, Berlin, 1987.

[14] Ernst Hairer and G. Wanner, *The Numerical Solution of Ordinary Differential Equations*, volume 2 (Stiff and Differential-Algebraic Problems), Springer-Verlag, Berlin, 1993.

[15] J. Hubbard and B. West, *Differential Equations, a dynamical systems approach*, volume 1, Springer-Verlag, 1991.

[16] Arieh Iserles, *private communication* 1991.

[17] W. Kahan, "Unconventional Numerical Methods for Trajectory Calculations", notes for a talk at the Fields Institute Workshop on Integration Algorithms for Classical Mechanics, Oct. 13–17, 1993.

[18] G. Labelle, *Sur l'inversion et l'iteration continue des séries formelles*, Europ. J. Combinatorics, **1** *pp.* 113–138, 1980.

[19] Rudolf J. Lohner, "Enclosing the solutions of ordinary initial and boundary value problems", in Proceedings of the 11th IMACS World Congres on System Simulation and Scientific Computation, Oslo, Volume 1, *pp.* 99–102, 1985.

[20] J. M. Sanz-Serna and M. P. Calvo, *Numerical Hamiltonian Problems*, Chapman and Hall, London, 1993.

[21] L. Shampine and M. Reichelt, "The Matlab ODE Suite", SIAM J. Sci. Comput., **81** no. 1, *pp.* 1–22, 1997.

[22] N. J. A. Sloane and S. Plouffe, *The encyclopedia of integer sequences*, Academic Press, 1994.

A Maple code

See `?libname` and `?savelib` to learn how to put this code in your local Maple library, so that it may be loaded in to your Maple session by the `readlib` command.

```
#
# Program to find the modified equation for
# y_{k+1} = y_k + h psi(y_k, t_k)
#
# MOA & RMC July 1997
#
# Input:
#    P  = psi = vector of expressions for
#          the method applied to the
#          differential equation already
#          with symbolic initial conditions
#          y_k = y(t).
#    y  = vector of unknown functions;
#          should be unassigned
#
#    n  = positive integer dimension of y
#    t  = integration variable
#    h  = time step (symbolic constant)
#    N  = desired order of computation.
#
#
#
```

```
modeq := proc( P::vector, y::vector, n::posint, t::name, h::name)
  local psi, sq, s, deriv, i, j, k, m, N;
  if nargs = 5 then
    N := Order-1
  else
    N := args[6]
  fi;
  s := vector(n,0);
  for k from 2 to N+1 do
    s := evalm(s +
         h^(k-1)/k!*vector(n, [seq( (D@@k)(y[j])(t), j=1..n)]) );
  od;
  psi := map(series, P, h, N+1);
  deriv := array(1..N+1);
  deriv[1] := map(series, evalm(psi - s), h, N+1);
  deriv[1] := map(convert, eval(deriv[1]), polynom);
  for i from 2 to N+1 do
    deriv[i] := vector(n,0);
    for k to n do
      deriv[i][k] := Normalizer(convert(diff(deriv[i-1][k],t),D));
    od;
  od;
  # At this point, deriv[N+1] contains only lower derivatives.
  for i from N by -1 to 1 do
    for j from i+1 to N+1 do
      deriv[i] := subs( seq( (D@@j)(y[k])(t)=deriv[j][k], k=1..n),
                        eval(deriv[i]) );
      deriv[i] := map(series, deriv[i], h, N+2-i);
      deriv[i] := map(convert, deriv[i], polynom);
    od;
    # At this point, deriv i contains only itself and lower derivs
    # We recursively remove deriv i, using O(log(N+1-i)) steps.
    m := 0;
    while 2^m <= N+1-i do
      deriv[i] := subs( seq( (D@@i)(y[k])(t)=deriv[i][k], k=1..n),
                        eval(deriv[i]) );
      deriv[i] := map(series, deriv[i], h, N+2-i);
      deriv[i] := map(convert, deriv[i], polynom);
      m := m+1;
    od; # while m
    # Now we remove deriv i from all higher derivs.
    for j from i+1 to N+1 while i > 1 do
      deriv[j] := subs( seq( (D@@i)(y[k])(t)=deriv[i][k], k=1..n),
                        eval(deriv[j]) );
      deriv[j] := map(series, deriv[j], h, N+2-j);
      deriv[j] := map(convert, deriv[j], polynom);
    od; # j
  od; # i
# The answer is just deriv[1].
eval(deriv[1])
end:
```