An Automatic Symbolic-Numeric Taylor Series ODE Solver*

Brian J. Dupée & James H. Davenport

Department of Mathematical Sciences, University of Bath, Claverton Down, Bath.
BA2 7AY. UK
email: {bjd,jhd}@maths.bath.ac.uk

Abstract. One of the basic techniques in every mathematician's toolkit is the Taylor series representation of functions. It is of such fundamental importance and it is so well understood that its use is often a first choice in numerical analysis. This faith has not, unfortunately, been transferred to the design of computer algorithms.

Approximation by use of Taylor series methods is inherently partly a symbolic process and partly numeric. This aspect has often, with reason, been regarded as a major hindrance in algorithm design. Whilst attempts have been made in the past to build a consistent set of programs for the symbolic and numeric paradigms, these have been necessarily multi-stage processes.

Using current technology it has at last become possible to integrate these two concepts and build an automatic adaptive symbolic-numeric algorithm within a uniform framework which can hide the internal workings behind a modern interface.

1 Introduction

This paper introduces a symbolic-numeric implementation of Taylor series methods for the solution of initial value ODE problems. Hitherto, the only implementations have been wholly numeric, wholly symbolic or obviously a multi-stage process (one where the symbolic and numeric calculations are carried out separately with at least one other stage between them). Such methods present a variety of computational problems, some of which are better solved symbolically, others numerically. This paper identifies the characteristics of such methods and presents an algorithm for better evaluation.

The techniques of approximating an ODE by a Taylor series has been known for many years and has been implemented, for example, by a group led by A. C. Norman in Cambridge, UK, in the package "TAYLOR" [11–13,2] using the analysis by Barton, Willers and Zahar [3]. An alternative package, 'ATSMCC' (Automatic Taylor Series by Morris, Chang and Corliss) [5], was also made available a few years later. In operation, these packages generate appropriate Fortran code to define the Taylor series and the evaluation process, given the ODE and initial conditions, for further compilation and operation. So they are at least a three-stage process. In [5], Corliss and Chang freely admit that:

When all the computer time for preprocessing, compiling, linking and execution was included, the relatively high system-dependent cost of linking with the library routines overwhelmed most other differences in CPU times.

The user-time performing the intervening stages (compilation, linking etc.) only added to the overall cost.

Some Computer Algebra Systems (CASs), such as Maple [8], Axiom [9] and Mathematica [15], have the necessary algorithms built-in to calculate the Taylor series symbolically. After all, this is a purely symbolic process. The evaluation stage is performed separately, usually symbolically even though this is computationally expensive.

The implementation in this paper, written for the CAS Axiom, is a single composite process which takes the ODE and initial conditions, creates the Taylor series, automatically generates the Fortran evaluation code which it then compiles, links and runs before returning the required result.

The trade-off for the speed of the evaluation process using Fortran is in the cost of the Fortran generation and compilation. It therefore makes sense to limit the size of the generated Fortran code as much as is reasonable. The speed-up over purely symbolic code is thus optimised. The better ease of use when compared to the purely numerical methods and applications like 'TAYLOR' and 'ATSMCC' is self-evident.

The paper includes comparisons between the new implementation, Taylor series methods using purely symbolic code and using alternative symbolic-numeric code.

2   Taylor Series Methods

The two papers by Barton, Willers and Zahar [2,3] provide a succinct and as complete a description and analysis of the method as are available. However, we will give a summary.

The method is described in [3] as an application of the process of analytic continuation. So, given a system of differential equations

$$F(t, y, y', y'', ..., y^{(n)}) = 0 \tag{1}$$

linear in $y^{(n)}$ and initial conditions

$$
\begin{aligned}
y_1 &= y(t_0) \\
y_2 &= y'(t_0) \\
y_3 &= y''(t_0) \\
&\;\;\vdots \\
y_n &= y^{(n-1)}(t_0)
\end{aligned}
$$

all specified, the Taylor series expansion about $t = t_0$ for y is given as

$$y_i(t) = \sum_{j=0}^{\infty} \frac{y_i^{(j)}(t_0)}{j!}(t - t_0)^j \tag{2}$$

The general idea is that this, or rather its approximation

$$y_i(t) = \sum_{j=0}^{N} \frac{y_i^{(j)}(t_0)\,h^j}{j!} \qquad (2')$$

is evaluated at $t = t_1 = t_0 + h$, the series is then expanded about this point and the process continued. The calculation of the Taylor series expansion is described in [2].

The analysis of the method in [3] shows that the method is extremely accurate and that, given a step-length $h = t - t_r$, the local error $E(h)$ will be small if

- the error involved in the truncation of the Taylor series is at least as small as the tolerance ϵ and
- the step-length chosen minimises $E(h)/h$.

The ideal truncation N is assumed to be the equal to the number of significant digits of the platform and is found to be only slightly dependent on the tolerance. However, since N was not found to be particularly sensitive to the equations integrated, this value must only be considered as an arbitrary ideal.

In the "TAYLOR" package, given the example

$$y''(t) - (1 - y(t)^2)y'(t) + y(t) = 0 \qquad (3)$$

(van der Pol's equation with $\mu = 1$) [16] and initial values

$$y(0) = 2$$
$$y'(0) = 0,$$

the Fortran code is generated using the commands (in a file)

```
INDEP T
DOUBLE
INIT SETUP
Y(0) = 2
Y'(0) = 0
EQNS
Y"=(1-Y**2)*Y' - Y
ADV VAL(Y,Y',T)
```

which will cause the program to create a number of Fortran subroutines for describing and evaluating the Taylor series. The user must then create a main program (in Fortran or C) to utilise these subroutines (enter parameters, print results etc.), compile, link and run the binary.

The main differences in 'ATSMCC' is in the calculation of the truncation value N and the appropriate step-size. Even though it requires less user control, the complete program has, in general, five stages:

1. create and edit the input,
2. preprocess to translate the input into object code,
3. compile the Fortran object code,

4. link with the ATSMCC subroutine library, and

5. execute the resulting load module to solve the problem.

Some of these steps may be combined and others may only need to be performed once if a single problem is to be solved repeatedly. However, this program structure does not lend itself towards a flexible and natural user-friendly interface.

The code supplied with some of the CASs use purely symbolic techniques. The main problems occur during the evaluation process since the number of substitutions (evaluations) of the parameters at each step is very large. Since substitutions are time-consuming, this part of the process is very inefficient. If the Taylor series evaluation process is compiled, there is some improvement, but not sufficient to boost the efficiency enough to be considered anything more than an exercise.

3 Composite Methods

Symbolic-numeric methods as implemented, for example, in ANNA [7] use symbolic analysis to identify sufficient parameters to be able to select and implement numeric procedures, usually from the NAG Fortran Library. The results are then returned for further symbolic processing. Underlying this is the ability within Axiom to create, compile and link Fortran evaluation procedures, named ASPs (Argument Sub-Programs), at run-time [4,10].

The technology to perform such processes, Nagman and nagd, is described in [6]. In essence, the Nagman local agent manages the data-transfer between the different computing protocols, using RPC (remote procedure call) to pass the data to the Nag daemon (nagd), which may be running on a remote system. The nagd incorporates a main program (stub), the Fortran library routine with appropriate C header files and the Axiom-generated ASP, performs the compilation and runs the resulting binary. The results are fed back through the Nagman to the current Axiom session.

The Fortran generation utilities included with recent implementations of Axiom are used to both create the Fortran code and also to verify that the full ANSI 1978 standards are adhered to with respect to variable names, types and constructs. The verification is necessary since the code may need to be compiled on a remote machine of indeterminate operating system and with any compiler.

All pre- and post-processing are performed within Axiom packages or the interactive session. The use of these packages, which may involve the use of the Axiom Category and Domain structures, allows for the automation of the complex processes and can thus become part of an intricate user interface.

In this implementation, since we are generating the algorithm, as opposed to the sub-programs, at runtime, the Fortran subroutine corresponding to the algorithm has to be compiled (using the same Fortran compiler as recognised by the nagd) and placed in a library to be accessed in a similar way to the access of the NAG Fortran Library's algorithms. The C header file is placed alongside the other C source files in the nagd directory structure and the Axiom code (the ASP Domain and the Taylor series method Package) is compiled as usual.

Algorithm 1: ODESolveTaylorSeries

% pre-processing

    calculate N from size of system

% symbolic-processing

    compute formal (lazy) Taylor series

    truncate Taylor series to order N

% Code creation, compilation and linking

    calculate Workspace requirements

    create ASP

    initiate Nagman

    compile Fortran code

    run Fortran code

% post-processing

    return result

    remove unwanted workspace

    print result

---

4   The Integrated Algorithm

The Taylor series representation is calculated symbolically within the Axiom Package `ExpressionSpaceODESolver` (written by Manuel Bronstein) which returns a lazy Taylor series in $n + 2$ variables where n is the order of the ODE. For this implementation this is truncated to form a polynomial in t of order $N = n + a$ where a is an integer ≥ 2 which can be estimated in the pre-processing stage (depending on the required accuracy and estimate of stiffness) and where the other variables may be of arbitrary order. So, for example, (3) could be approximated by (in the case where $a = 2$):

$$\left(-\frac{1}{12} y_3^3 + \left(\frac{1}{3} y_2^3 - \frac{1}{3} y_2 \right) y_3^2 \right.$$

$$\left. + \left(-\frac{1}{24} y_2^6 + \frac{1}{8} y_2^4 + \frac{5}{24} y_2^2 - \frac{1}{24} \right) y_3 - \frac{1}{24} y_2^5 + \frac{1}{12} y_2^3 \right) t^4$$

$$+ \left(\frac{1}{3} y_1 y_3^3 + \left(-\frac{4}{3} y_1 y_2^3 + \left(\frac{4}{3} y_1 - \frac{1}{3} \right) y_2 \right) y_3^2 \right.$$

$$+ \left(\frac{1}{6} y_1 y_2^6 + \left(-\frac{1}{2} y_1 + \frac{1}{6} \right) y_2^4 + \left(-\frac{5}{6} y_1 - \frac{1}{3} \right) y_2^2 + \frac{1}{6} y_1 \right) y_3$$

$$\left. + \frac{1}{6} y_1 y_2^5 + \left(-\frac{1}{3} y_1 + \frac{1}{6} \right) y_2^3 - \frac{1}{6} y_2 \right) t^3$$

$$+ \left(-\frac{1}{2} y_1^2 y_3^3 + \left(2 y_1^2 y_2^3 + \left(-2 y_1^2 + y_1 \right) y_2 \right) y_3^2 + \left(-\frac{1}{4} y_1^2 y_2^6 \right.\right.$$

$$+ \left(\frac{3}{4} y_1^2 - \frac{1}{2} y_1 \right) y_2^4 + \left(\frac{5}{4} y_1^2 + y_1 - \frac{1}{2} \right) y_2^2 - \frac{1}{4} y_1^2 + \frac{1}{2} \right) y_3$$

$$\left. - \frac{1}{4} y_1^2 y_2^5 + \left(\frac{1}{2} y_1^2 - \frac{1}{2} y_1 \right) y_2^3 + \left(\frac{1}{2} y_1 - \frac{1}{2} \right) y_2 \right) t^2$$

$$+ \left(\frac{1}{3} y_1^3 y_3^3 + \left(-\frac{4}{3} y_1^3 y_2^3 + \left(\frac{4}{3} y_1^3 - y_1^2 \right) y_2 \right) y_3^2 + \left(\frac{1}{6} y_1^3 y_2^6 \right.\right.$$

$$+ \left(-\frac{1}{2} y_1^3 + \frac{1}{2} y_1^2 \right) y_2^4 + \left(-\frac{5}{6} y_1^3 - y_1^2 + y_1 \right) y_2^2 + \frac{1}{6} y_1^3 - y_1 + 1 \right) y_3$$

$$\left. + \frac{1}{6} y_1^3 y_2^5 + \left(-\frac{1}{3} y_1^3 + \frac{1}{2} y_1^2 \right) y_2^3 + \left(-\frac{1}{2} y_1^2 + y_1 \right) y_2 \right) t$$

$$- \frac{1}{12} y_1^4 y_3^3 + \left(\frac{1}{3} y_1^4 y_2^3 + \left(-\frac{1}{3} y_1^4 + \frac{1}{3} y_1^3 \right) y_2 \right) y_3^2$$

$$+ \left(-\frac{1}{24} y_1^4 y_2^6 + \left(\frac{1}{8} y_1^4 - \frac{1}{6} y_1^3 \right) y_2^4 + \left(\frac{5}{24} y_1^4 + \frac{1}{3} y_1^3 - \frac{1}{2} y_1^2 \right) y_2^2 \right.$$

$$\left. - \frac{1}{24} y_1^4 + \frac{1}{2} y_1^2 - y_1 \right) y_3 - \frac{1}{24} y_1^4 y_2^5 + \left(\frac{1}{12} y_1^4 - \frac{1}{6} y_1^3 \right) y_2^3 +$$

$$\left(\frac{1}{6} y_1^3 - \frac{1}{2} y_1^2 + 1 \right) y_2 \quad (4)$$

The calculation of the optimum value of the truncation parameter N is critical. This is since the prime costs in this algorithm are in the Fortran generation, compilation and linking stages, as opposed to the evaluation stages of a fully symbolic system or the purely numeric stages of "TAYLOR". We can therefore consider, if necessary, using smaller step-sizes than those recommended in [3] rather than a larger polynomial system to cut down this cost but we then leave ourselves open to problems with any ODE showing more than the mildest of stiffness (since in a stiff system, there will be more information contained in the coefficients of the higher exponents of the Taylor series)[1]. We must, though, ensure that all appropriate y^i are calculated.

It must be noted that the step-sizes mentioned above are, or can be, much larger than the step-sizes used in other ODE solvers such as Runge-Kutta. The initial 'test evaluation' can use a step-size of as much as $\frac{1}{10}$ of the complete range of integration. So the usual number of steps is often much less than other methods.

The polynomial is then coded into a Fortran ASP. Each monomial becomes one line of Fortran code to better facilitate the calculation of the return values y, y' ... y^n (see Appendix A.1). This is then passed to the nagd for compilation and linking.

---

[1] An alternative Taylor series algorithm for stiff equations is described in [1] but this is much more expensive computationally and it is not apparent that there would likely be any major cost or accuracy benefits over other stiff methods.

Other pre-processing stages include the estimation of the amount of workspace required for the calculation stages and the setting up of the appropriate matrices. For this, a reasonable estimate must be made of the number of iterations that may be required in the calculation. This is not an easy task since the algorithm is designed to be adaptive i.e. alter the step-length according to the changes in the values of y, y' etc. Obviously a 'quick and dirty' method of estimating the workspace requirements may be inaccurate but sufficient. A reasonable overestimate is all that is required. This is calculated as a function of the truncation parameter, N, the order of the ODE, n, and the required tolerance.

The Fortran library program (see Appendix A.2), after checking input values for consistency, calculates a weight function (a function of the input values) and evaluates its first step. On the result of this evaluation it re-calculates the weight function to get a good basic estimate for the step-size before recalculating the first step. It then goes through the evaluation process modifying the step-size as appropriate. All of the calculated values for y, y' ... y^n are returned.

In order to present the information in as consistent a way as possible, the post-processing stages remove some of the excess workspace and re-order those that are left. It does, however, one other important job. If the numerical process was unable to get a sufficient answer, it may re-calculate the initial step-size or increase the amount of workspace before re-initiating the calculation.

5   Examples

Example 1: Using the system, (3) is solved with the commands:

```
y := operator 'y;
eq := D(y(t),t,2) - (1 - y(t)^2)*D(y(t),t) + y(t) = 0
solve(eq,y,t=0..20,[2,0],0.01)$TaylorSeriesODENumPackage
```

which calculates the values of y and y' from $t = 0$ to $t = 20$ and gives the result:[2]

```
(4)
[ifail: Integer, result: Matrix(DoubleFloat),
 y: Matrix(DoubleFloat), xout: Matrix(DoubleFloat),
 yin: Matrix(DoubleFloat), count: Integer]
                                    Type: Result
```

---

[2] The returned fields in the result are:

ifail : The return status — a non-zero value indicates a failure
y : The values of y, y' ... y^n at the end point
result : The values of y, y' ... y^n at all iteration points corresponding to xout (for display etc.)
xout : The iteration points and the end point (for display etc.)
yin : the values of y, y' ... y^n at the last iteration point
count : the number of iterations.

```
(5) -> %.'y
```

```
(5)  [2.00897152959078   - 0.0370573647342098]
                            Type: Matrix DoubleFloat
```

in 214 iterations. The field y here contains the values of y and y' at the end point.

If the value of μ is set in the van der Pol equation to increase the stiffness i.e. (3) is changed to:

$$y''(t) - 5(1 - y(t)^2)y'(t) + y(t) = 0 \qquad (5)$$

then this is mildly stiff and 4 d.p. accuracy can be achieved without increasing the truncation parameter, N, at the cost of increasing the number of iterations to 647. Any further increase in stiffness would require an increase in N.

Example 2: The ODE

$$y'(t) + y(t) = \sin t \qquad (6)$$

with initial conditions

$$y(0) = 1$$

is calculated from $t = 0$ to $t = 20$ using the commands:

```
eq2 := D(y(t),t) + y(t) = sin(t)
solve(eq2,y,t=0..20,[1],0.1)$TaylorSeriesODENumPackage
```

and this produces:

```
(6)
[ifail: Integer, result: Matrix(DoubleFloat),
 y: Matrix(DoubleFloat), xout: Matrix(DoubleFloat),
 yin: Matrix(DoubleFloat), count: Integer]
                                    Type: Result
(7) -> %.'y
```

```
(7)  [0.252446658993222]
                            Type: Matrix DoubleFloat
```

in 176 iterations.

The same examples were solved using the Taylor series method incorporated within Maple and using Runge-Kutta or Adams methods as appropriate within ANNA both as the result only and with 200 intermediate results. The new algorithm and the Maple symbolic algorithm were also timed including plotting the resulting graph. The timings are as follows[3]:

---

[3] All tests were carried out on a SUN Sparc Classic named 'dictum' under Solaris 2.5.1 at the University of Bath using Axiom 2.2 and Maple V release 4.00f to 4 d.p.

Table 1. Sample test timings

	Example 1	Example 2
Maple (Taylor)	197s	14s
including plot	795s	153s
ANNA (result only)	37s	7s
intermediate results	65s	56s
New Algorithm	66s	14s
including plot	72s	16s

Commentary: The above results require some explanation. The increase in time needed for 200 intermediate results using ANNA is due to the smaller step-size required. The Maple results are affected by the difficulty to tailor results to a given precision but similar results are obtained with a purely symbolic algorithm written in Axiom.

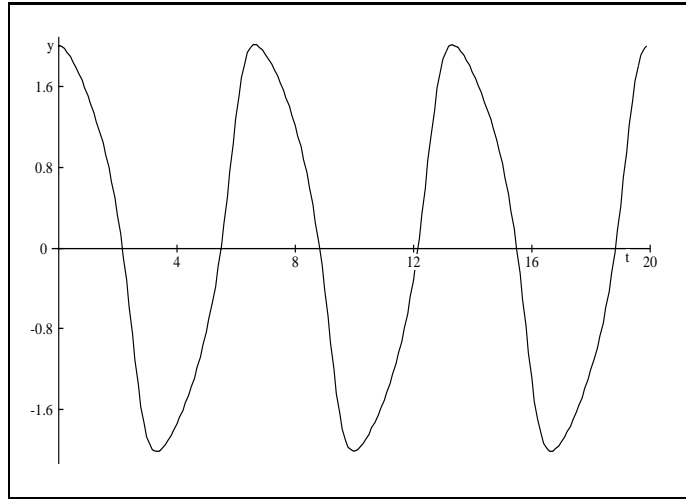A simple display function provided with the package can then produce:

Fig. 1. van der Pol's equation with $\mu = 1$, $y(0) = 2$ and $y'(0) = 0$.

6   Conclusion

Tests show considerable speed-up over purely symbolic processing, except for the smallest of systems, but actual figures differ due to different platform/platform combinations. However, the single most immediate effect is of the simple interface and the amount of information returned. This extra information is achieved at little extra cost.

With a non-stiff system, as (3), the accuracy of the system is remarkable. If a variable loading is applied to affect the stiffness, the accuracy drops dramatically

unless the order of polynomial approximation is increased. But this affects the time involved in the Fortran generation stages. This means that for anything more than mild stiffness, the algorithm is not optimal.

For simple systems, the cost can be comparable to other symbolic-numeric methods as implemented within ANNA, although the main benefits are in terms of the achieved accuracy and whether intermediate results are required for, say, display. However, this method is not appropriate to those ODEs for which the CAS cannot find the recurrence relation.

Further work will eventually include the incorporation of this method into ANNA. For this, a measure function will be created which calculates the system size, stiffness and stability of the ODE and returns a value which corresponds to the ability of this particular method to solve the ODE efficiently. It is clear from the above that increasing stiffness will warrant a larger polynomial approximation and therefore reduce the measure, up to a point where a different method (such as BDF) would be more appropriate. Similarly a larger system and decreasing stability would have the same effect.

It will also be possible under certain circumstances to increase the range of ODEs which can be solved by this method, such as where in (1) F is not linear in $y^{(n)}$ but a linear system can be calculated by differentiation. This could be done within the pre-processing stages of the algorithm.

References

1. BARTON, D. On taylor series and stiff equations. *ACM Trans. Math. Softw. 6*, 3 (Sept. 1980), 281–294.
2. BARTON, D., WILLERS, I. M., AND ZAHAR, R. M. V. The automatic solution of systems of ordinary differential equations by the method of taylor series. *Computer Journal 14*, 3 (1971).
3. BARTON, D., WILLERS, I. M., AND ZAHAR, R. M. V. Taylor series methods for ordinary differential equations — an evaluation. In Rice [14], pp. 369–390.
4. BROUGHAN, K. A., KEADY, G., ROBB, T., RICHARDSON, M. G., AND DEWAR, M. C. Some symbolic computing links to the NAG numeric library. *SIGSAM Bulletin 25* (July 1991), 28–37.
5. CORLISS, G. F., AND CHANG, Y. F. Solving ordinary differential equations using taylor series. *ACM Trans. Math. Softw. 8*, 2 (June 1982), 114–144.
6. DEWAR, M. C. Manipulating fortran code in AXIOM and the AXIOM-NAG link. In *Workshop on Symbolic and Numeric Computation (1993)* (Helsinki, 1994), H. Apiola, M. Laine, and E. Valkeila, Eds., pp. 1–12. Research Report B10, Rolf Nevanlinna Institute, Helsinki.
7. DUPÉE, B. J., AND DAVENPORT, J. H. An intelligent interface to numerical routines. In *DISCO'96: Design and Implementation of Symbolic Computation Systems* (Karlsrühe, 1996), J. Calmet and J. Limongelli, Eds., vol. 1128 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, pp. 252–262.
8. HECK, A. *Introduction to Maple*, 2nd ed. Springer Verlag, New York, 1996.
9. JENKS, R. D., AND SUTOR, R. S. *AXIOM: The Scientific Computation System*. Springer-Verlag, New York, 1992.
10. KEADY, G., AND NOLAN, G. Production of argument subprograms in the AXIOM-NAG link: examples involving nonlinear systems. In *Workshop on*

Symbolic and Numeric Computation, May 1993 (Helsinki, 1994), H. Apiola, M. Laine, and E. Valkeila, Eds., pp. 13–32. Research Report B10, Rolf Nevanlinna Institute, Helsinki.

11. NORMAN, A. C. *TAYLOR Users Manual*. Computing Laboratory, University of Cambridge, UK, 1973.

12. NORMAN, A. C. Computing with formal power series. *ACM Trans. Math. Softw. 1* (1975), 346–356.

13. NORMAN, A. C. Expanding the solutions of implicit sets of ordinary differential equations. *Comp. J 19* (1976), 63–68.

14. RICE, J. R., Ed. *Mathematical Software*. Academic Press, New York, 1971.

15. WOLFRAM, S. *The Mathematica Book*, 3rd ed. CUP, Cambridge, 1996.

16. ZWILLINGER, D. *Handbook of Differential Equations*, 2nd ed. Academic Press, San Diego, CA, 1989.

A Fortran Code

A.1 ASP

```
 SUBROUTINE EVAL(N,PREC,XT,YIN,YVAL,YOUT)
 INTEGER PREC,N
 DOUBLE PRECISION XT,YIN(N)
 DOUBLE PRECISION YOUT(PREC),YVAL(PREC)
 YVAL(1)=(-0.08333333333333333 D0*YIN(1)**4*YIN(3)**3)+(0.3333333
&333333333 D0*YIN(1)**4*YIN(2)**3+((-0.3333333333333333 D0*YIN(1)
&**4)+0.3333333333333333 D0*YIN(1)**3)*YIN(2))*YIN(3)**2+((-0.041
&66666666666666 D0*YIN(1)**4*YIN(2)**6)+(0.125D0*YIN(1)**4+(-0.16
&66666666666667 D0*YIN(1)**3))*YIN(2)**4+(0.2083333333333333 D0*Y
&IN(1)**4+0.3333333333333333 D0*YIN(1)**3+(-0.5D0*YIN(1)**2))*YIN
&(2)**2+(-0.04166666666666666 D0*YIN(1)**4)+0.5D0*YIN(1)**2+(-1.0
&D0*YIN(1)))*YIN(3)+(-0.04166666666666666 D0*YIN(1)**4*YIN(2)**5)
&+(0.08333333333333333 D0*YIN(1)**4+(-0.1666666666666667 D0*YIN(1
&)**3))*YIN(2)**3+(0.1666666666666667 D0*YIN(1)**3+(-0.5D0*YIN(1)
&**2)+1.0D0)*YIN(2)
 YVAL(2)=0.3333333333333333 D0*YIN(1)**3*YIN(3)**3+((-1.333333333
&333333 D0*YIN(1)**3*YIN(2)**3)+(1.333333333333333 D0*YIN(1)**3+(
&-1.0D0*YIN(1)**2))*YIN(2))*YIN(3)**2+(0.1666666666666667 D0*YIN(
&1)**3*YIN(2)**6+((-0.5D0*YIN(1)**3)+0.5D0*YIN(1)**2)*YIN(2)**4+(
&(-0.8333333333333334 D0*YIN(1)**3)+(-1.0D0*YIN(1)**2)+YIN(1))*YI
&N(2)**2+0.1666666666666667 D0*YIN(1)**3+(-1.0D0*YIN(1))+1.0D0)*Y
&IN(3)+0.1666666666666667 D0*YIN(1)**3*YIN(2)**5+((-0.33333333333
&33333 D0*YIN(1)**3)+0.5D0*YIN(1)**2)*YIN(2)**3+((-0.5D0*YIN(1)**
&2)+YIN(1))*YIN(2)
 YVAL(3)=(-0.5D0*YIN(1)**2*YIN(3)**3)+(2.0D0*YIN(1)**2*YIN(2)**3+
&((-2.0D0*YIN(1)**2)+YIN(1))*YIN(2))*YIN(3)**2+((-0.25D0*YIN(1)**
&2*YIN(2)**6)+(0.75D0*YIN(1)**2+(-0.5D0*YIN(1)))*YIN(2)**4+(1.25D
&0*YIN(1)**2+YIN(1)-0.5D0)*YIN(2)**2+(-0.25D0*YIN(1)**2)+0.5D0)*Y
&IN(3)+(-0.25D0*YIN(1)**2*YIN(2)**5)+(0.5D0*YIN(1)**2+(-0.5D0*YIN
&(1)))*YIN(2)**3+(0.5D0*YIN(1)-0.5D0)*YIN(2)
 YVAL(4)=0.3333333333333333 D0*YIN(1)*YIN(3)**3+((-1.333333333333
&333 D0*YIN(1)*YIN(2)**3)+(1.333333333333333 D0*YIN(1)-0.33333333
&33333333 D0)*YIN(2))*YIN(3)**2+(0.1666666666666667 D0*YIN(1)*YIN
&(2)**6+((-0.5D0*YIN(1))+0.1666666666666667 D0)*YIN(2)**4+((-0.83
&33333333333334 D0*YIN(1))-0.3333333333333333 D0)*YIN(2)**2+0.166
&6666666666667 D0*YIN(1))*YIN(3)+0.1666666666666667 D0*YIN(1)*YIN
&(2)**5+((-0.3333333333333333 D0*YIN(1))+0.1666666666666667 D0)*Y
&IN(2)**3+(-0.1666666666666667 D0*YIN(2))
 YVAL(5)=(-0.08333333333333333 D0*YIN(3)**3)+(0.3333333333333333
&D0*YIN(2)**3+(-0.3333333333333333 D0*YIN(2)))*YIN(3)**2+((-0.041
&66666666666666 D0*YIN(2)**6)+0.125D0*YIN(2)**4+0.208333333333333
&3 D0*YIN(2)**2-0.04166666666666666 D0)*YIN(3)+(-0.04166666666666
&666 D0*YIN(2)**5)+0.08333333333333333 D0*YIN(2)**3
 YOUT(1)=YVAL(5)*XT**4+YVAL(4)*XT**3+YVAL(3)*XT*XT+YVAL(2)*XT+YVA
&L(1)
```

```
      YOUT(2)=4.0D0*YVAL(5)*XT**3+3.0D0*YVAL(4)*XT*XT+2.0D0*YVAL(3)*XT
     &+YVAL(2)
      YOUT(3)=12.0D0*YVAL(5)*XT*XT+6.0D0*YVAL(4)*XT+2.0D0*YVAL(3)
      YOUT(4)=24.0D0*YVAL(5)*XT+6.0D0*YVAL(4)
      YOUT(5)=24.0D0*YVAL(5)
      RETURN
      END
```

A.2 Library Subroutine

```
      SUBROUTINE TAYLOR(M,N,PREC,COUNT,X0,XEND,XOUT,YIN,
     &                  WORK,EVAL,EVALOUT,YOUT,TOL,IFAIL)

      INTEGER M,N,COUNT,PREC,IFAIL
      DOUBLE PRECISION X0,XEND,TOL
      DOUBLE PRECISION XOUT(M),YIN(N),WORK(PREC)
      DOUBLE PRECISION YOUT(N,M),EVALOUT(PREC)
C     Local variables
      INTEGER I
      DOUBLE PRECISION H,W,NEXT,STEP
      LOGICAL BACKWARD, FORWARD

      EXTERNAL EVAL
      INTRINSIC ABS

C     Check values of input variables for consistency
      IF ((TOL.LE.0).OR.(XEND.EQ.X0).OR.(PREC.LE.N))
      THEN
        IFAIL = 1
        GOTO 202
      ENDIF
      YIN(1)=X0

C     Put initial value in output
      COUNT=1
      DO 49 I=1,N-1
        YOUT(I,1)=YIN(I+1)
49    CONTINUE
      XOUT(COUNT)=X0

C     First try of stepsize is 1/10th of distance
      H=(XEND-X0)/10.0
C     Divided by a weight function
      W=0.0D0
      DO 50 I=1,N
        W=W+2.0D0*ABS(YIN(I))
50    CONTINUE
      STEP=H/(1.0D0+W/N)
      NEXT=X0+STEP
```

```
C     Call the evaluate subroutine
      CALL EVAL(N,PREC,NEXT,YIN,WORK,EVALOUT)
100   IF (ABS(EVALOUT(1)-YIN(2)).GT.TOL) THEN
C        Recalculate H to get a better stepsize
         H=H/2.0
         STEP=H/(1.0D0+W/N)
         NEXT=X0+STEP
         CALL EVAL(N,PREC,NEXT,YIN,WORK,EVALOUT)
         GOTO 100
      ENDIF

C     Increment
      COUNT=COUNT+1

C     Put result in output
      DO 51 I=1,N
         YOUT(I,COUNT)=EVALOUT(I)
51    CONTINUE
      XOUT(COUNT)=NEXT


      BACKWARD=(XEND.LT.X0)
      FORWARD=.NOT.BACKWARD

C     While not finished
200   IF ((FORWARD.AND.(NEXT.LT.XEND)).OR.
     &         (BACKWARD.AND.(NEXT.GT.XEND))) THEN
C        Increment
         IF (COUNT.GE.M) THEN
            IFAIL=2
            GOTO 202
         ENDIF
         COUNT=COUNT+1
C        Result of last evaluation is input for the next
         YIN(1)=NEXT
         DO 52 I=2,N
            YIN(I)=EVALOUT(I-1)
52       CONTINUE
C        Calculate new weight from all taylor precision
         W=0.0D0
         DO 53 I=1,PREC-1
            W=W+ABS(EVALOUT(I))
53       CONTINUE
C        Calculate new step
         STEP=H/(1.0D0+W/(PREC*PREC))
         IF (STEP.EQ.0) THEN
            IFAIL=3
            GOTO 202
         ENDIF
```

```fortran
          NEXT=NEXT+STEP
C         If we've got to the end then make it the end
          IF ((FORWARD.AND.(NEXT.GT.XEND)).OR.
     &         (BACKWARD.AND.(NEXT.LE.XEND))) THEN
            NEXT=XEND
          ENDIF
C         Call then new evaluation
          CALL EVAL(N,PREC,NEXT,YIN,WORK,EVALOUT)
C         Put in output
          DO 54 I=1,PREC
            YOUT(I,COUNT)=EVALOUT(I)
54        CONTINUE
          XOUT(COUNT)=NEXT
          GOTO 200
C         End while loop
        ENDIF
        IFAIL=0
201     CONTINUE

202     RETURN
        END
```