The automatic solution of systems of ordinary differential equations by the method of Taylor series

D. Barton, I. M. Willers and R. V. M. Zahar

Computer Laboratory, University of Cambridge, Cambridge, England

A method is described for the automatic reduction of an arbitrary system of ordinary differential equations to a form suitable for solution by the method of Taylor series. The paper then describes how a program may be generated from the reduced form that will carry out the formal solution of the system by that method. Further it is shown that the reduction and the program generation may be performed by a program that is similar in many respects to a compiler that compiles programs written in a simple language. This language is defined in the paper. The method has been implemented and several examples are presented.

(Received August 1970)

The Taylor series method for the solution of a system of ordinary differential equations is well known. However, the difficulty of implementing a general-purpose algorithm that will enable an arbitrary differential system to be solved by this method is so great that the procedure has been largely neglected in the literature in favour of computationally more simple techniques. Briefly, the method is to replace each variable present in the differential system by a Taylor series centred at a certain origin. With the exception of the constant terms the coefficients in each series are regarded as unknown quantities. The differential equations of the system may be used to obtain a set of recurrence relations from which these unknowns can be calculated. Thus a formal power series solution may be determined to an initial value problem and the series will be convergent in some region about the origin. The truncated formal series are then evaluated at some point within the region and hence a new set of initial values obtained. The recurrence relations then yield a second series solution valid in a region about the new origin and consequently the differential equations may be solved by a numerical version of the technique of analytic continuation into a finite region. We shall not be concerned with the numerical techniques that are available for choosing the new origin as this has been discussed by Barton, Willers and Zahar (1970), but here confine ourselves to the automatic derivation of the recurrence relations and their efficient encoding.

There is a systematic approach to the problem that is frequently employed by hand. The equations of the differential system are first reduced to a certain canonical form from which the recurrence relations may be easily deduced. These relations are then written down explicitly and finally an efficient integration procedure is constructed that employs the recurrence relations. The procedure of reduction to canonical form and the derivation of the recurrence relations is best illustrated by an example. Consider the differential system

\[ y' = y^2 + z \]
\[ z' = z^2 \]

where

\[ y = z = 1 \text{ when } t = 0 \]

Introducing new variables \( a, b, c \), we obtain the canonical form for system (1)

\[ a = y^2 \]
\[ b = a + z \]
\[ c = z^2 \]
\[ y' = b \]
\[ z' = c \]

To obtain a formal series solution to system (2) about the point \( t = t_0 \) we write

\[ y = \sum_{i=0}^{\infty} y^{(i)}(t-t_0)^i \]

together with similar expansions for the variables \( z, a, b \) and \( c \). Then from the equations of system (2) we obtain the recurrence relations

\[ a^{(N)}_0 = \sum_{j=0}^{N} y^{(N-j)}_0 b^{(j)}_0 \]
\[ b^{(N)}_0 = a^{(N)}_0 + z^{(N)}_0 \]
\[ c^{(N)}_0 = \sum_{j=0}^{N} z^{(N-j)}_0 c^{(j)}_0 \]
\[ y^{(N+1)}_0 = b^{(N)}_0(N+1) \]
\[ z^{(N+1)}_0 = c^{(N)}_0(N+1) \]

while the initial conditions of system (2) yield \( y^{(0)}_0 = z^{(0)}_0 = 1 \), \( t_0 = 0 \).

A complication arises when the original differential system contains non-rational functions for example \( \sin(x) \) or \( \cos(x) \). It is possible to deal with these functions in two different ways, and we illustrate both techniques by reference to the example of the system

\[ y' = 1 + \sin(y) \]

where

\[ y = 1 \text{ when } t = 0. \]

It is known (for example, see Moore, 1966) that the Taylor coefficients of \( \sin(y) \) may be determined by a recurrence relation when those of \( y \) are known. With this result in mind a suitable canonical reduction of system (4) is given by (5) on introducing new variables \( a, b \).

\[ a = \sin(y) \]
\[ b = 1 + a \]
\[ y' = b \]

where

\[ y = 1 \text{ when } t = 0. \]

The appropriate recurrence relations may now be directly written down.

To illustrate the second technique let us assume that the recurrence relation for the Taylor coefficients of \( \sin(y) \) in terms of those of \( y \) was not known. In this case we introduce
the new variables $a$ and $b$ and augment system (4) by the differential equations defining $\sin$ and $\cos$. Hence we obtain

$$
\begin{align*}
  y' &= 1 + a \\
  a' &= by' \\
  b' &= -ay'
\end{align*}
$$

where

$$
  y = 1, \quad a = \sin (1), \quad b = \cos (1) \text{ when } t = 0.
$$

Introducing new variables $c, d, e, f$ and $g$ we obtain from system (6) the canonical system

$$
\begin{align*}
  c &= 1 + a \\
  d &= bc \\
  e &= ac \\
  f &= -e \\
  y' &= c \\
  a' &= d \\
  b' &= f
\end{align*}
$$

where

$$
  y = 1, \quad a = \sin (1), \quad b = \cos (1) \text{ when } t = 0.
$$

We see therefore that non-rational functions may be treated either by the use of a recurrence relation when this is available or alternatively by the use of the defining differential equations. However, the latter technique complicates the problem to some extent and, of course, raises the order of the differential system.

The general procedure described above for the solution of an initial value problem is clearly a tedious and error-prone task for even a moderately complicated set of equations. A number of programming systems have been produced that automate parts of the general method, but for all of the programs it is necessary to present the differential system in canonical form. For equations presented in this form Gibbon (1960) describes an implementation that treats non-rational functions by both of the techniques described above while Moore (1966) describes two systems that treat non-rational functions by the two methods separately.

In this paper we show that an initial value problem, stated in a formal but natural manner, may be considered as a sentence of a simple language. The syntactic analysis of this sentence yields a tree structure from which the canonical form of the differential equations of the problem may be obtained. Further we show that an optimised form of the canonical representation may be deduced from the initial representation and finally that a program, in any suitable target language, corresponding to the equivalent recurrence relations (cf. equation (7)) used in an appropriate order may be obtained. The entire process is analogous to the generation of code from a sentence written in a programming language and the program that carries it out will be referred to as a compiler. The compiled program may then be called by a numerical system for the integration of the differential equations. The complete system has been implemented on the Atlas 2 computer in Cambridge. The Atlas implementation is an interactive program and Fig. 1 is a sample protocol for the solution of an initial value problem that is self-explanatory.

The automatic reduction of a differential system to standard form

In our discussion of the syntactic analysis and compilation of an initial value problem we omit any description of the statements controlling the output of numerical values during the integration procedure and accordingly assume that the PRINT statement in the example, Fig. 1, is replaced by the word END. The general form of presentation of an initial value problem to the system is therefore

INTEGRATE

$$
y_i^{(n)} = f_i(t, y_{1}, \ldots, y_{i-1}, \ldots, y_{m}, \ldots, y_{m+n-1}, r_{1}, \ldots, r_{p})
\quad 1 \leq i \leq m
$$

where

$$
r_i = g_i(t, y_{1}, \ldots, y_{i-1}, \ldots, y_{m}, \ldots, y_{m+n-1}, r_{1}, \ldots, r_{i-1},
\quad r_{i+1}, \ldots, r_{p})
\quad 1 \leq i \leq p
$$

WITH INITIAL CONDITIONS

$$
y_i^{(j)} = h_i(y_{1}, \ldots, y_{i-1}, \ldots, y_{m-1})
\quad 1 \leq i \leq m \quad 0 \leq j < n_i - 1
$$

$\quad t = t_{0}$

END

where $k_i$ and $l_i$ are positive integers and in the equation for $y_i^{(n)}$ we have $k_i < n_i$.

Further $t$ is the independent variable. It must be understood that for the purpose of input to the computer each equation defines the variable on the left-hand side explicitly and that for our implementation distinct variables $y_i$ and $r_i$ are written using different letters of the alphabet. Further, we write

$$
y_i^{(j)} = \frac{dy_i}{dt}
$$

as a letter followed by a sequence of $j$ primes. Finally, we call the equation defining $y_i^{(n)}$ a differential equation and that defining $r_i$ an identity.

Consider now the language composed of the following basic symbols where composite symbols are underlined:

$$
+ \quad - \quad * \quad \uparrow \quad = \quad \text{INT} \quad \text{NL}
$$

```
INTEGRATE WITH INITIAL CONDITIONS END
```

and defined by the grammar

<initial value problem> \rightarrow INTEGRATE \text{NL} <differential system> WITH INITIAL CONDITIONS \text{NL} <initial point> END

<initial value problem> \rightarrow <equation> | <equation> <differential system>

<equation> \rightarrow <name> = <expression> \text{NL} | <name> = \text{INT} <expression> \text{NL}

<initial value problem> \rightarrow <value> | <value> <initial point> \rightarrow <name> = <expression> \text{NL}

Here the operator \text{INT} is diadic and $y = \text{INT} <expression>$ means integrate the expression, assign the result to $y$ and yield the result $y$. $A <name>$ is the address of a portion of space allocated to a certain variable appearing in the differential system and a $<\text{numerical constant}>$ is a suitably marked address of the location of the value of that constant. Finally an $<\text{expression}>$ is an arithmetic expression composed of $<\text{names}>$ and $<\text{numerical constants}>$ with the usual operators $+-*\uparrow\text{neg sin cos etc}$, together with the parentheses $( )$ in the usual way, except that we allow the operator $\text{prime}$ (prime) as well. The operator $'$ denotes differentiation with respect to $t$. 

![Fig. 1](https://academic.oup.com/comjnl/article-abstract/14/3/243/420218/0)
This language plays a central role in the compilation that proceeds, in four stages, to reduce the source text of an initial value problem to a program in target language (e.g. machine code or FORTRAN). This program will solve the problem by the method of Taylor series. It should be mentioned that this grammar will accept ill-defined as well as well-defined initial value problems and the detection of badly posed problems is left until the code generation stage of compilation. The four distinct stages of compilation are as follows:

1. Lexical analysis.
2. Formal syntax analysis and reduction to matrix form.
3. Optimisation of the matrix form.

These will be described in turn with reference to the example

```
INTEGRATE
Y' = 2R + SIN (Y + 1) + 1 + Y
R = (1 + Y) \uparrow 2
WITH INITIAL CONDITIONS
Y = Y'
Y' = 1
T = 0
END
```

1. **Lexical analysis**

The purpose of this stage of compilation is to translate the source text of the initial value problem into a sentence in the language defined above. It is logically a prepass of the source but for the Cambridge implementation it appears as part of the input subroutine. However, at this first stage it is convenient to perform certain other manipulations of the source that greatly simplify the syntax analysis carried out at stage two. Consequently, while processing a differential equation or an identity, the analyser operates in two modes. In mode one, before reading the equals sign, the only sequence that is accepted is `<letter>` `<possibly null sequence of primes>`.

This sequence causes the analyser to assign names for the several variables that will be required during the later work, thus the sequence `Y'` causes the names `Y''`, `Y'` and `Y` to be assigned for the quantities `Y'', Y'` and `Y` respectively. Then, for a differential equation, where `n` is the number of primes occurring, the sequence `<letter>` `<n primes>` =, is translated to the sequence of basic symbols.

```
<name for letter> = \int <name for letter'> = \int ... while, for an identity, the sequence <letter> =, is translated to the basic symbols <name for letter> =.

Thus the sequence `Y''` = is translated to `Y = \int Y' = \int Y' = \int` and the sequence `Y =` is translated to `Y =`,. Having read the equals sign the analyser changes to mode two in which it accepts any sequence of characters and recognises particularly the sequence `<letter>` `<primes>` and translates it to the appropriate name followed by the minimum sequence of prime operators. In mode two, numerical constants are also recognised and compiled, their translated value being their names. Finally, in mode two an explicit multiplication operator `*` is inserted between any two names found in juxtaposition and also, where appropriate, between parentheses and names. Thus the lexical analyser translates the initial value problem given above into

```
INTEGRATE n1 Y = \int Y' = \int 2 \ast R \ast \sin (Y + 1) + 1 + \\
Y n1 R = (1 + Y) \uparrow 2 n1 \\
WITH INITIAL CONDITIONS n1 Y = Y' n1 Y' = 1 n1 \\
T = 0 n1 END
```

where names and numerical constants are denoted by the corresponding underlined symbols.

2. **Formal syntax analysis and reduction to matrix form**

This stage of compilation reads the output from stage one and translates it first to a syntax tree and then condenses this to an equivalent matrix form. It is this matrix form that is equivalent to the canonical form of the differential system from which the recurrence relations for the Taylor series method may be deduced. Algorithms for the reduction to a syntax tree and to matrix form are well known and we do not give a detailed description of them here (see Graham, 1964). First, those parts of the syntax tree that refer to initial conditions and second, those parts that refer to the identities and the differential equations are independently reduced to matrix form. The first of these two forms is later used to generate an initialisation program and the second is used to generate the program to calculate Taylor series solutions of the initial value problem by means of recurrence relations. In Fig. 2 is shown the syntax tree for our example and Fig. 3 shows the two matrix forms.

3. **Optimisation of the matrix form**

During the third and fourth stages of compilation the two matrix forms are treated independently; however, the same optimisation procedure is used for both. Clearly the most fundamental form of optimisation is the recognition and elimination of common subexpressions. This is of particular importance for the recurrence matrix (the matrix form of the differential equations) as it is from there that a program for the recurrence procedure is generated, and any redundancy there will have very serious effects on the runtime of the final compiled program. Common subexpressions are therefore detected by direct matching of the rows of the matrix and also by use of the commutative law for addition and multiplication followed by matching. The optimiser continues to work on the matrix until it fails to improve the representation in one entire pass through the matrix. The second type of optimisation that takes place is the elimination of unnecessary operations between Taylor series. Thus the product of a constant and a series is marked as such and can be separately treated at the code
generation stage. We show the optimised matrix form in Fig. 4 and operators between constants and series have been enclosed in circles. Finally, arithmetic between constants is carried out and the corresponding rows of the matrix are deleted.

4. Code generation

The optimised matrix form of the recurrence matrix and the initialisation matrix that results from stage three of the compilation is simply a representation of the canonical form of the initial value problem and it remains to generate from it a program in the target language for use during the numerical integration. However, this task is not straightforward since the original equations may not have been presented in an order suitable for immediate evaluation and also because algorithms for the various elementary functions may not be available. In our example (Fig. 4(a), if an algorithm is available to evaluate the \( n \)th Taylor coefficient of \( \sin(x) \) when both of \( x \) and \( \sin(x) \) are known up to the \( (n-1) \)th coefficients and the \( n \)th coefficient of \( x \) is known, then the matrix form can be transcribed as it stands. However, if such an algorithm is not available then the system must append rows to both matrices (Fig. 4(a), (b)) to correspond to the differential equations defining \( \sin(x) \) and their initial conditions respectively. The code generator is provided with a list of elementary functions together with either the algorithm or the appropriate differential equations and deals with them according as either is known. For the present we assume that an algorithm is known and hence that the matrices (Fig. 4(a), (b)) can be processed without modification.

The code generator constructs three segments of program. The first of these is the initialisation program and this is used just once to initialise the variables for the first step of the integration. For subsequent steps the variables are initialised by summing the previously generated Taylor series. To obtain the initialisation program the code generator scans the rows of the initialisation matrix until it finds a row corresponding to an operation that can be performed immediately. If the search is successful then program appropriate to that row is generated, the row is removed and the search continues. The program is complete when the matrix is totally deleted. If the procedure fails to terminate or if some of the variables defined by differential equations are not suitably initialised then the problem is not well posed and the system rejects it. In our case the matrix (Fig. 4(b)) results in the following program being generated where, for convenience, we represent each variable by a vector whose elements are the coefficients of the Taylor series representing that variable and the target language is FORTRAN. The name \( YP \) represents \( Y'. \)

\[
YP(1) = 1.0 \\
Y(1) = YP(1) \\
T(1) = 0.0
\]

It is of course found that our problem is well posed.

The other two segments of program that are generated are necessary to implement the recurrence procedure that will enable all the Taylor coefficients to be calculated in turn. One of these segments of program, the prologue, is concerned with the evaluation of sufficient terms in each Taylor series to enable the final recurrence procedure to be started. The other segment, the recurrence loop, simply advances each series by one term and continues until sufficient terms have been calculated in each Taylor series to give an accurate numerical representation of the associated function. At each step of the integration the prologue is first obeyed and then the recurrence loop completes the generation of the new Taylor series.

The need for the prologue is perhaps not immediately obvious and indeed for many simple differential systems it is null. However, if the differential operator \( f \) (prime) is present on the right-hand side of a differential equation or identity then the need for a prologue becomes apparent. Consider the equation

\[
y' = y + (\sin^2 t)^r
\]

Before we can determine the first term of the Taylor series for \( (\sin^2 t)^r \) we must, in effect, compute the first three terms of the series for \( \sin^2 t \). For this implementation it is the job of the prologue to calculate the first two of these and that of the recurrence loop to complete the evaluation of the series for \( y \).

Before translating the recurrence matrix into its corresponding prologue and recurrence loop, it is obviously necessary to devise an effective ordering of the operations. As we have seen, this problem is complicated by the possible occurrence of differentiation operators in the recurrence matrix. Thus, for each dependent variable \( Y_i \) or each auxiliary variable \( R_i \) (including in both cases those that are introduced by the compiler) it is necessary to discover the other \( Y_i \) and \( R_i \) on which it directly depends. Once this is done, it is necessary to calculate, for a fixed number of terms in the series for the former variable, the number of terms needed in the series for the latter variables. We note that the auxiliary variables \( R_i \) on which a variable depends need to be known to one higher term than the dependent variables \( Y_i \) because the former do not occur as the result of an integration operation. An algorithm for deciding the relative number of terms needed in each series and for generating the prologue and recurrence loop is now described formally.

Let \( \{Z_i\} \) be the set of variables that occur as the results of all operations in the recurrence matrix. Let \( \{Y_i\} \) be the subset of \( \{Z_i\} \) which are the results of an integration and \( \{R_i\} \) be such that \( \{Z_i\} = \{Y_i, R_i\} \). Consider the matrix \( D \) that contains a labelled row and column for each \( Z_i \). We regard the rows of the matrix \( D \) as corresponding to the operations in the recurrence

\[
\begin{array}{ccc}
\text{OPN} & \text{ARG 1} & \text{ARG 2} \\
\times & 2 & Y' \\
\div & 1 & T2 \\
+ & 1 & T4 \\
\div & 1 & T5 \\
= & 1 & T6 \\
\div & 2 & T7 \\
\end{array}
\]

Fig. 3a. Recurrence matrix

\[
\begin{array}{ccc}
\text{OPN} & \text{ARG 1} & \text{ARG 2} \\
\times & 2 & Y' \\
\div & 1 & T2 \\
\end{array}
\]

Fig. 3b. Initialisation matrix

\[
\begin{array}{ccc}
\text{OPN} & \text{ARG 1} & \text{ARG 2} \\
\times & 2 & R \\
\div & 1 & T2 \\
\end{array}
\]

Fig. 4a. Optimised recurrence matrix

\[
\begin{array}{ccc}
\text{OPN} & \text{ARG 1} & \text{ARG 2} \\
\times & Y' & T4 \\
\div & 1 & T5 \\
\end{array}
\]

Fig. 4b. Optimised initialisation matrix
matrix that define the \( Z_i \). To construct \( D_{ij} \) we first select the row of the recurrence matrix which has the result \( Z_i \). Then we trace the arguments of this operation back through the recurrence matrix following each branch of the tree independently and stopping when we reach any \( Y \) or any constant or the independent variable \( i \). Let \( n \) be the number of differentiation operators encountered en route and provided that \( D_{ij} \) is increased by the operation, set \( D_{ij} = n \) when a \( Y \) is encountered and \( D_{ij} = n + 1 \) when an \( R \) is encountered. Otherwise \( D_{ij} \) is set to negative infinity and can be omitted from further consideration.

Now let \( C \) be the set of integers \( i \) for which \( D_{ij} \leq 0 \) for all \( j \). Thus, row \( i \) corresponds to an operation resulting in \( Z_i \) which can be used immediately. Let \( M \) be the set of integers \( j \) such that \( D_{ij} \geq 2 \) for some \( i \), so that variable \( Z_i \) must be computed in the prologue in part. If \( C \cap M \neq \emptyset \) choose the greatest \( i \in C \cap M \) and compile program that corresponds to the operation yielding \( Z_i \). This program is part of the prologue.

Now, since \( D_{ij} \) can be regarded as the difference between the number of terms needed for \( Z_i \) and the number obtained by \( Z_j \), we set \( D_{ij} = D_{ij} + 1 \) and \( D_{ij} = D_{ij} - 1 \) for all \( j \). Then we recompute the sets \( C \) and \( M \) and repeat the procedure until \( C \cap M = \emptyset \) which occurs in a finite number of steps because no \( D_{ij} > 0 \) is even increased and some \( D_{ij} \geq 2 \) is decreased on each step. When \( C \cap M = \emptyset \) we have either \( M = \emptyset \) or \( M \neq \emptyset \) and \( C \cap M = \emptyset \). In the latter case the problem is not well-posed since no operation can be used that will result in those variables that must be computed in the prologue.

Consider therefore the case \( M = \emptyset \) and \( C \neq \emptyset \). We now generate the third segment of the program that is the recurrence loop. Choose the greatest \( i \in C \) and generate program that corresponds to the operation resulting in \( Z_i \), delete the row \( i \) from \( D \) and set \( D_{ij} = D_{ij} - 1 \) for all \( j \). Then recompute \( C \) and continue until either \( C = \emptyset \) or the matrix \( D \) is null. If \( C = \emptyset \) before \( D \) is null, the problem is not well posed and if \( D \) becomes null, the program for the recurrence loop is complete. It is of course understood that the program that forms part of the prologue and recurrence loop must be suitably indexed and the recurrence loop itself must be controlled by an appropriate count.

In the case of our example, the matrix \( D \) is

\[
\begin{bmatrix}
T1 & T2 & T3 & T4 & Y' & Y & R \\
T1 & 1 & 0 & 1 & \\
T2 & 0 & & & \\
T3 & 1 & 0 & & \\
T4 & 1 & 1 & 0 & \\
Y' & 1 & 1 & 1 & 1 & 0 & 1 \\
Y & 1 & 1 & 1 & 1 & 1 & 0 \\
R & 0 & 1 & 0 & \\
\end{bmatrix}
\]

The prologue is therefore null and the program for the recurrence loop is shown in Fig. 5.

Let us now consider the more complicated example of the equations

\[
Y'' = (XR)'' \\
R = XX \\
X' = Y
\]

The optimised recurrence matrix is

<table>
<thead>
<tr>
<th>Opn</th>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>X</td>
<td>R</td>
<td>T1</td>
</tr>
<tr>
<td>'</td>
<td>T1</td>
<td></td>
<td>T2</td>
</tr>
<tr>
<td>'</td>
<td>T2</td>
<td></td>
<td>T3</td>
</tr>
<tr>
<td>=j</td>
<td>Y'</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>=j</td>
<td>X</td>
<td>Y</td>
<td>R</td>
</tr>
<tr>
<td>=j</td>
<td>X</td>
<td>Y</td>
<td>X</td>
</tr>
</tbody>
</table>
DO 5 I=1,N
C THE INTEGRATION OF Y'
Y(I+1)=Y(I)+FLOAT(I)/FLOAT(I+1)
C THE INTEGRATION OF Y
X(I+2)=Y(I+1)/FLOAT(I+1)+1.0
C THE EVALUATION OF R=XX
R(I+2)=0.0
DO 3 J=1,4
3 R(I+2)=R(I+2)+X(I+2)*X(J)
C THE EVALUATION OF RX
T(I+2)=0.0
DO 4 J=1,4
4 T(I+2)=T(I+2)+R(I+3-J)*X(J)
C THE DIFFERENTIATION OF (RX)
T2(I+2)=T(I+2)*T2(I+2)+T(I+2)*T(I+2)
C THE DIFFERENTIATION OF (RX^2)
T3(I+2)=FLOAT(I+2)*T3(I+2)
C THE INTEGRATION OF (RX)^2
Y(I+2)=T3(I+2)/FLOAT(I+2)
C END OF LOOP
5 CONTINUE

which yields the recurrence loop shown in Fig. 7.

That completes our description of the compiler and the reduction of a differential system to canonical form. We now discuss some points that have a marginal effect on the efficiency of the compiled program. We have, throughout the paper, referred to a <numerical constant> as simply a floating-point number identified by the input routine, and we have seen that operations between these quantities may be carried out at compile time and further, that arithmetic between them and bona fide Taylor series may, in certain cases, be carried out by a simpler program than that required for two Taylor series. It will, no doubt, have occurred to the reader that some of these remarks are also true of functions of the independent variable t. It is an open question whether it is better to treat, let us say, sin(t) as a function whose Taylor series about a given point $\varphi_0$ can be calculated by explicit techniques completely in the prologue or to allow it to be generated by recurrence relations by simply adding (internally) the equation $t' = 1$ with the appropriate initial condition, and so making the system autonomous. For the Cambridge implementation the latter view is adopted for simplicity.

The runtime speed of the compiled program is also to some extent affected by the representation of $<\text{numerical constants}>$ and known functions of the independent variable. If they are represented as full Taylor series they are obviously wasteful of space while the alternative is that more orders must be obeyed at runtime if another representation is chosen. Consider the operation $Y + t$ occurring in the recurrence loop. It is clear that for terms in the Taylor series of degree higher than one the operation need not be performed at all, and the function $t$ need only be stored as two numbers. However, this would mean that the generated program must contain suitable tests to ensure that the operation of addition is carried out only for two terms and these tests could be omitted if the function $t$ were represented as a full Taylor series filled out with zeros. The choice to be adopted here will depend mainly on the target language and for the Cambridge implementation we have chosen the former alternative, the target language being Atlas machine code.

The efficiency of the system can be further improved by optimal treatment of non-rational functions. In our implementation we have found that some of these (sin, cos) are more efficiently generated from the defining differential equations than from the recurrence relation, but this may well not be true for them all. However, there is no doubt that the exponentiation operator must be carefully optimised to repeated multiplication whenever this is desirable.

Once the compiler has generated an efficient program that will discover a Taylor series solution to an initial value problem given the initial point, the speed of the step by step integration procedure is determined by the numerical techniques used to choose the step size and to ensure accurate evaluation of the generated series. These techniques, together with an evaluation of the Taylor series method, have been discussed in Barton, Willers and Zahar (1970).

References


