The syntax directed graph algorithm for the input of equations to the Taylor series system for solving ordinary differential equations

I. M. Willers

Cern Laboratoire 1, 1211 Genève 23, Switzerland

A new algorithm is described for the automatic generation of code for the Taylor series method of solving ordinary differential equations. The equations are represented by 'syntax directed graphs'. It is demonstrated that this is a natural development from the classical syntax tree. A compiling algorithm is then described which, when applied to this structure, generates object code which may be used for generating Taylor series by the use of appropriate recurrence relations. Finally this method is compared with the algorithm in Barton, Willers and Zahar, 1971.

(Received September 1974)

In Barton, Willers and Zahar (1971) it is described how the Taylor series method for the solution of the initial value problem of ordinary differential equations may be implemented as a completely automatic procedure.

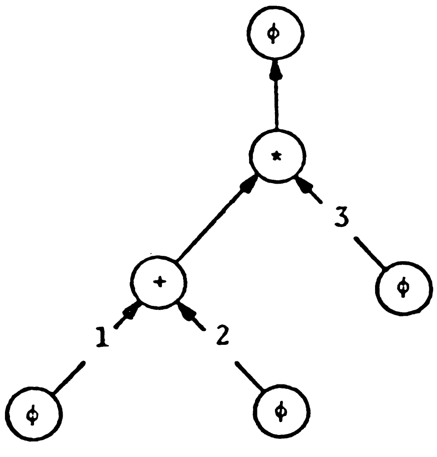
In this paper an alternative description of the compiling process and code generation stages of the implementation are given. As well as being easier to describe and to understand, this new description leads to a simpler implementation of the algorithm. The numerical aspects of the Taylor series method are covered in Barton, Willers and Zahar (1970). The basic method is described in Moore (1966) and outlined in the two papers by Barton, Willers and Zahar. In short, the unknown variables of the equations are replaced by truncated Taylor series. A recurrence relationship between the Taylor series coefficients is established using the equations themselves. The equations are split up into a set of simple equations which involve one arithmetic operation and at least one unknown variable which may have been introduced at this stage. Each new variable has a Taylor series associated with it. Each one of these simple equations corresponds to a relationship between the respective Taylor coefficients. This relationship depends upon the arithmetic operation, and may be simply stated. These simple relationships, when ordered, produce the recurrence relationship. The recurrence relationships and the equation's initial conditions may be used to obtain a normal truncated power series solution to the problem. Using suitable numerical control, the dependent variables may then be evaluated at another point, hence giving new initial conditions, and the process can be repeated to give a complete step-by-step procedure.

This paper introduces the concept of a 'syntax directed graph' (SDG for short), which is a generalisation of the syntax tree normally used when compiling arithmetic expressions (see Graham, 1964).

The SDG structure is used both to establish that a problem may be well posed, and for the generation of the recurrence relationships in the form of object code.

1. Construction of the syntax directed graph

Let us consider a directed graph. We let the vertices represent arithmetic operators and a null operator, ϕ, (e.g. $+$, $-$, $*$, $/$, ϕ, . . .), and let the arcs represent variables. Then, the expression $(1 + 2) * 3$ may be mapped onto a directed graph as follows:

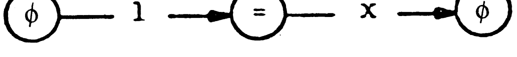


This structure is directly analogous to a syntax tree, the construction of which is well understood (see Graham, 1964).

However, in the Taylor series system the objective is to input sets of equations. Let the equals sign, $=$, be added to the set of operators and consider the equation

$$x = 1$$

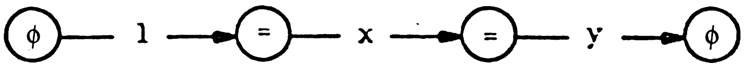
mapping this onto an SDG gives



We can add another equation to the set and obtain

$$x = 1$$
$$y = x$$

which maps onto



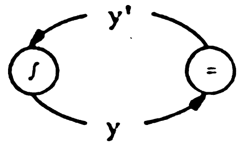
Thus two equations have mapped onto a single SDG and the relationship between the two equations is clearly indicated. The process of adding an equation to a set may be seen to be a simple process which may involve the deletion of null vertices.

Consider the differential equation:

$$y' = y$$

where $'$ on the left hand side implies integration of the right hand side.

This maps on to the SDG

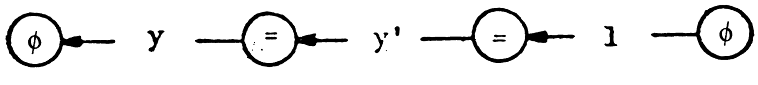


This introduces the interesting occurrence of a loop in the SDG. The structure of a syntax tree is completely lost and the relationship expressed in the equation cannot be expressed in a syntax tree. In this way a problem statement for the Taylor series system may be mapped onto two sets of SDG's, one set representing the differential system and the other set the initial values.

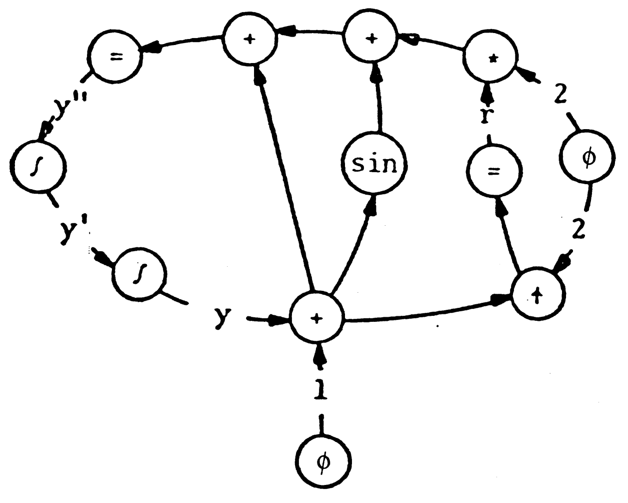
Consider the example:

Integrate
$y'' = 2 * r + \sin(y + 1) + 1 + y$
$r = (1 + y) ** 2$
with initial conditions
$y = y'$
$y' = 1$

The initial conditions map onto



The two equations form one connected SDG which indicates how one of the equations must be used before solving the other. The differential equation and the identity map onto



The SDG's can be optimised by a straightforward comparison of vertices and this process is made easier, as far as implementation is concerned, if the variables, or arcs, pointing into the vertices are ordered. The removal of common subexpressions is clearly demonstrated in the example.

Each prime that appears on the left hand side of the equation leads to an integration of the right hand side. The looping properties of the SDG are clearly shown.

2. The compiling algorithm

One task of this algorithm is to determine whether a problem statement may be well defined for the Taylor series algorithm.

A typical problem statement has the form:

Integrate
$y_i^{(n_i)} = f_i(t, y_1, y_2, \ldots, y_m) \quad 1 \leqslant i \leqslant m$
with initial conditions
$y_i^{(j)} = h_i^j(t, y_1, y_2, \ldots, y_m) \quad 1 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant n_i$
$t = t_0$

where (j) and (n_i) represent a nonnegative number of differentiations and t is the independent variable.

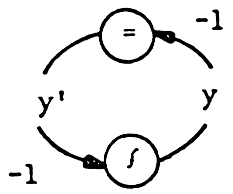
We express the two sets of equations as two sets of SDG's and define a labelling of the SDG which enables the algorithm to be applied. Each variable is represented as a truncated Taylor series and each operation is a relationship acting on these series, producing a single result which is a coefficient of a series. Consider such an operation on j Taylor series where the ith Taylor series is known up to and including the coefficient of order m_i. Let this operation produce a coefficient of order n. Then there exists a general relationship of the form:

$$c \leqslant n \leqslant \min(m_1, m_2, \ldots, m_j) + c \ , \qquad (1)$$

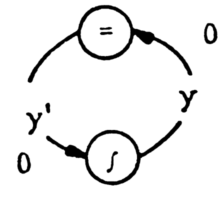
where c is a function of the operation such that $c = 1$ for ∫, $c = -1$ for ′ and $c = 0$ otherwise. For example, if one wanted to add two series together and one series is known up to order 3 whilst the other is known to order 5, then the sum of these two series can be calculated up to order 3. Clearly, for addition, $c = 0$.

Let the value -1 and the orders of known coefficients be associated with each series. Each variable in a problem statement, or equivalently, each arc of an SDG has associated with it a set of integers. Using the relationship deduced above, the action of an operation can be simulated by assigning a set of integers to the set of arcs leaving a vertex. For a given vertex, which represents an operation, it is possible to associate the set $[c, c + 1, \ldots, n]$ (defined by the relationship (1)) with the set of arcs leaving the vertex. Let us now consider only sets of the type $[-1, 0, 1, \ldots, n]$, and characterise this set by the use of the integer n. An integer, bounded by the relationship (1), can be assigned to each set of arcs in an SDG.

Consider the example of a differential equation $y' = y$.



The value -1 is given to each arc, and will be referred to as the value of that arc. Relationship (1) allows the association of the value zero with y, but $c > 0$ for integration so this operation is not possible. Simulation of the operator '=' produces a similar situation. If, however, the initial value of arc y is made zero, it is possible to simulate the equals operation to give the following picture:



Now the action of the ∫ operation can be simulated to give the value 1 to arc y. This process may be continued so that with each arc is associated an integer which is as large as is desired. It should be realised that giving the value zero to arc y is equivalent to giving the differential equation an initial value, without which it is impossible to solve.

Theorem:
If, at the beginning, it is possible to raise the value of each finite valued arc by one, then it is possible to raise the values of each finite valued arc to an integer which is as large as is desired.

Proof:
Consider an arc whose finite value can be raised by one. Let its value be raised by one to n, which is bounded by the relationship (1)

$$c \leqslant n \leqslant \min(m_1, m_2, \ldots, m_j) + c$$

Now let the values of every other finite valued arc be raised by one. Then, in particular, the value of every finite m_i has changed. Now we have

$$c \leqslant n + 1 \leqslant \min(m_1 + 1, m_2 + 1, \ldots, m_j + 1) + c$$

thus it is possible to raise the value of the arc which is under consideration. Since this is true for every finite arc, then all of their values can be raised to an integer which is as large as is desired.
QED

The aim of the algorithm is to raise the value of each arc by one. The simulation of this particular sequence of operations corresponds to a simulation of the recurrence relationship.

The algorithm

The SDG's of the problem statement are drawn, the value infinity is assigned to arcs which represent constants or the independent variable when known to order zero. The value -1 is assigned to the remaining arcs.

The algorithm is applied to the initial conditions. If this is successful the values of the arcs in the SDG of the initial conditions are copied to similarly named arcs in the SDG of the differential equations. Then the algorithm is repeated for the SDG of the differential equations.

The nodal application of the algorithm is the act of raising the value of an arc by one to the value n using the relationship

$$c \leqslant n \leqslant \min(m_1, m_2, \ldots, m_j) + c \ .$$

A single application of the algorithm consists of a series of nodal applications. After a nodal application the arc to which a value is assigned is marked (by a tick, say), and subsequent nodal applications are made to unmarked arcs until this becomes impossible, thus finishing a single application. When there is no unique order of nodal applications the order may be chosen arbitrarily.

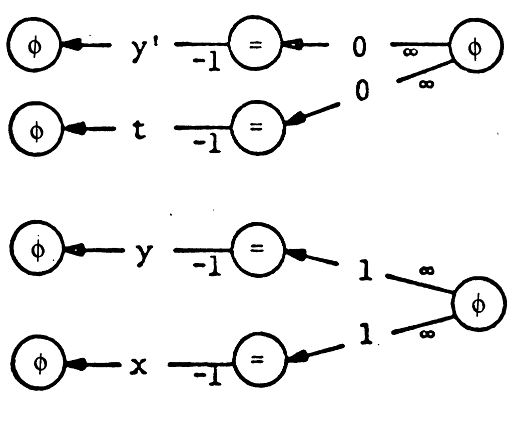
A single application is made and, if all arcs are ticked, the set of equations may be well posed. If no previously unmarked arc was marked by a tick the set of equations is ill posed. Otherwise the arcs that are marked are recorded, the ticks removed and the algorithm repeated.

The algorithm halts in a finite number of steps, since either on each single application an arc is marked for the first time, or else the process is halted, and there are a finite number of arcs.

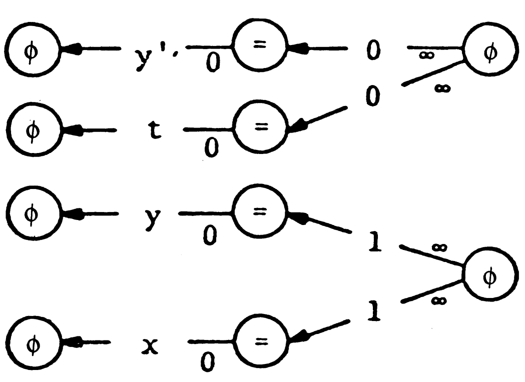
Consider the application of the algorithm to the example:

Integrate
$y'' = (x * r)''$
$r \ = x * x$
$x' = y$
with initial conditions
$y \ = 1$
$y' = 0$
$x \ = 1$
$t \ = 0$

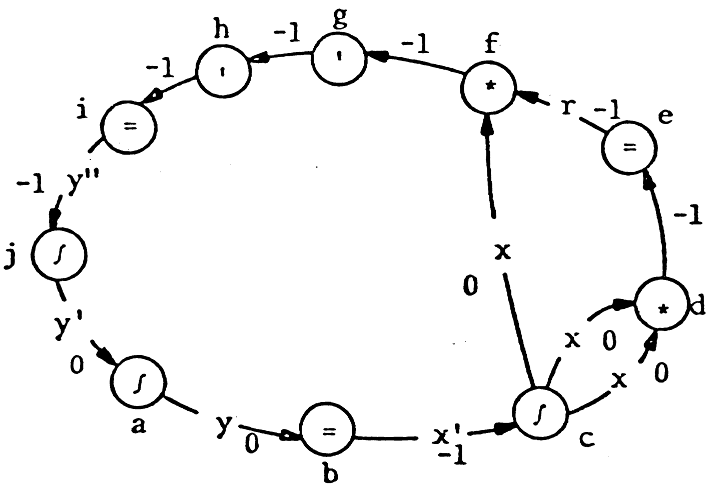
The SDG's for the initial conditions are drawn and labelled.



The nodal applications are made in any order giving:



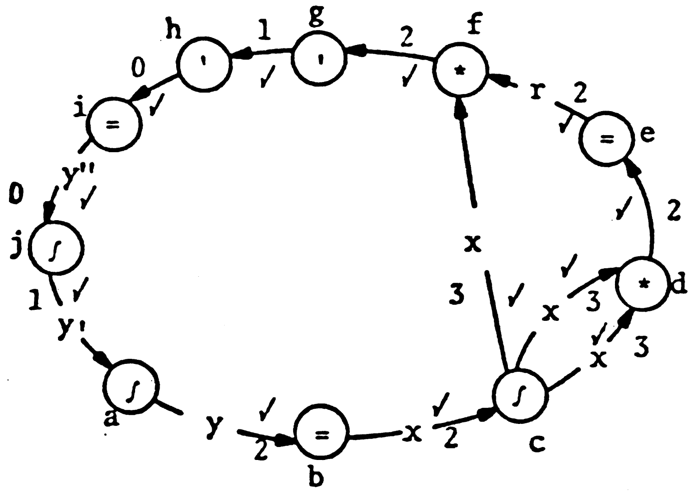
The initial conditions are well posed, and so the SDG for the differential equations is drawn and labelled thus



In this example it is interesting to note that all the arcs have finite values. Initially, nodal applications can be made at the vertices (*a*), (*b*) or (*d*). Let the algorithm be applied to vertices (*a*), (*b*), (*c*), (*d*), (*e*) and (*f*) in that order.

Since it is impossible to continue, this completes a single application. Again, there is a choice about the order in which nodal applications can be made. The algorithm is applied to vertices (*b*), (*c*), (*d*), (*e*), (*f*) and (*g*).

The procedure is repeated but now the order of nodal applications is unique and their application gives:



All the arcs are marked by ticks, and so the problem may be well posed.

The output of code
Each nodal application of the algorithm can be thought of as a simulation of the action of the operational relationship at a vertex. Therefore as a nodal application is made, the appropriate object code is generated. If the code that is being generated is from the last single application it is surrounded by a programmed loop. This code within the loop corresponds to a recurrence relationship and the whole loop is known as the recurrence loop. For most efficient object code it may be necessary to scan ahead to ascertain that a single application is the final one. For the above example object code is written as a series of sentences which describe the operations. The operations act upon truncated series and produce a single result which is the coefficient of a series with the given order. A series of comments are written down the right hand side of the column to indicate the corresponding nodal applications.

The subroutine for the initial conditions is:

```
                 Set coefficient order = 0
                 Set $y' = 0$
                 Set $y = 1$
                 Set $x = 1$
                 Set $t = 0$
                 End
```

The differential equations lead to the code

```
Comment: beginning of first single application
                 Set coefficient order = 0
                 Integrate $y'$ to $y$ $(=x')$                (a, b
                 Integrate $x'$ to $x$                        (c
                 Set $t_1 = x * x (=r)$                       (d, e
                 Set $t_2 = x * r$                            (f
Comment: beginning of second single application
                 Set coefficient order = 1
                 Integrate $x'$ to $x$                        (c
                 Set $t_1 = x * x (=r)$                       (d, e
                 Set $t_2 = x * r$                            (f
                 Differentiate $t_2$ to $t_3$                 (g
Comment: beginning of third single application
                 Set coefficient order = 2
Loop commences:  Set $t_1 = x * x (=r)$                       (d, e
                 Set $t_2 = x * r$                            (f
                 Differentiate $t_2$ to $t_3$                 (g
Decrement coefficient order
                 Differentiate $t_3$ to $t_4$ $(=y'')$        (h, i
Decrement coefficient order
                 Integrate $y''$ to $y'$                      (j
Increment coefficient order
                 Integrate $y'$ to $y$ $(=x')$                (a, b
Increment coefficient order
                 Integrate $x'$ to $x$                        (c
Increment coefficient order and loop until required order is
achieved
                 End
```

3. The relationship between the new and existing algorithms

The new algorithm may be considered to be a simpler version of the existing algorithm given in Barton, Willers and Zahar (1971). For those wishing to compare the two algorithms, the following is intended to summarise the precise relationship between them.

Syntax analysis
New algorithm The SDG is formed in a similar manner to a syntax tree. Optimisation takes place as the SDG is being constructed.

Existing algorithm A syntax tree is constructed, and is then converted to matrix form. Each row of the matrix is equivalent to the classical canonical form as described in Moore (1966). As the matrix form is constructed the rows are searched for possible optimisations.

Comparison The optimised SDG is entirely equivalent to the optimised matrix form.

Code generation, first stage
New algorithm A number is associated with each arc of the SDG. This number represents the highest order of known Taylor series coefficients for each variable. Constants and the independent variable are recognised. The code for the initial conditions is generated and the SDG for the differential equations modified accordingly.

Existing algorithm A matrix D is constructed which contains information regarding the difference between the number of terms needed and the number obtained for a particular operation. This involves tracing back through the matrix form whilst counting differentiation operators, and recognising constants, the independent variable and variables which are the result of an integration. Code for the initial conditions is generated.

Comparison The matrix D, the matrix form and the orders of the coefficients that have been dealt with are equivalent to the SDG. By tracing back through the SDG, it is possible to generate the matrix D.

Code generation, second stage
New algorithm The algorithm is applied to the nodes of the SDG, and code is generated as the algorithm progresses.

Existing algorithm The matrix D is scanned and appropriate code is generated using the matrix form and stored information on the order of the coefficients that have been dealt with.

Comparison The SDG contains all the required information; however, it may be necessary to scan ahead to deduce that this is the final application. The matrix D may be examined to ascertain the beginning of the recurrence loop. The code that is produced should be of a similar quality.

References

Barton, D., Willers, I. M., and Zahar, R. V. M. (1971). The automatic solution of systems of ordinary differential equations by the method of Taylor series. *The Computer Journal*, Vol. 14, No. 3, pp. 243-248; also in *The Best Computer Papers of 1971*, Auerbach (Ed.) Petrocelli, Philadelphia, 1971, pp. 147-163.

Barton, D., Willers, I. M., and Zahar, R. V. M. (1970). Taylor series methods for ordinary differential equations—an evaluation, *Proc. Math. Software Symposium*, Purdue University, Lafayette, Indiana.

Graham, R. M. (1964). Bounded context translation, *AFIPS SJCC*, Vol. 25, p. 17-29.

Moore, R. E. (1966). *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ.