Software for the numerical solution of systems of functional differential equations with state-dependent delays

K.W. Neves
Computational Mathematics, Boeing Computer Services, Seattle, WA 98124, USA

S. Thompson
Radford University, Radford, VA 24145, USA

Abstract


The theoretical basis for the numerical solution of a general class of functional differential equations is reviewed. A software package for the solution of differential equations with state-dependent delays is discussed. The package uses continuously imbedded Runge-Kutta methods of Sarafyan. These methods are based on $C^1$ piecewise polynomial interpolants which are used to handle tasks associated with root finding and interpolation. In addition to providing a means to handle user-defined root finding requirements, they provide a means to locate automatically derivative discontinuities that arise in the solution of differential equations with delays. Examples are presented which demonstrate the manner in which the software takes into account the pertinent theoretical characteristics of functional differential equations.

1. Introduction

This paper is concerned with the solution of systems of functional differential equations (FDEs) which contain delays or lags. The techniques discussed are applicable to a broad class of delay differential equations that includes constant delays, strictly time-dependent delays, and more generally, state-dependent delays (SDLs). The methods used in the software which is described provide a natural means of solving such problems. Readers interested in the question of other software for problems with delays are referred to [24-28,31] and the references therein.

Standard initial value techniques are not directly applicable to delay problems since evaluation of the derivative at time $t$ requires the solution and/or the derivative at previous times. However, standard techniques may be modified and used to solve delay problems. This approach was used in the DMRODE delay solver discussed in [25,26]. DMRODE proved to be very successful and has been widely used. A primary objective of the present work was to use...
many of the mathematical ideas which formed the basis for DMRODE along with more recent theoretical understanding of FDEs and develop a delay solver which incorporates important advances in ODE software technology (e.g., improvements in root finding, error estimation, step size selection, and interpolation techniques). This approach takes advantage of the wealth of accumulated knowledge for standard ODE software. Thus, we adopted the approach of using a standard integration method to advance the integration, saving the solution as it is generated, and employing some type of interpolation to approximate the solution at previous times.

A key feature of any software for handling delay equations is the method of interpolation. For many methods, (see [11]) the order of interpolation must be at least $p$ for a $p$th order method, for this will guarantee local error of order $p + 1$ (which is necessary to maintain global error of order $p$). Some methods provide a natural and consistent solution for the interpolation problem. For example, linear multistep methods such as the Adams or backward differentiation methods used in Gear-type solvers (e.g., the SDRIV2 solver discussed in [21]) are based on polynomial interpolants. These interpolants may be used to approximate the delayed variables. Since the solution of the differential equation is also based on the polynomial interpolants, the delayed variables are approximated as accurately as the integrated solution. In fact, the adaptation of one of the best available nonstiff multistep solvers has been recently reported. However, a drawback of multistep methods is the difficulty of advancing the solution in a neighborhood of a low-order derivative discontinuity. Since this is a common occurrence with delay equations, Runge–Kutta methods are also popular for the solution of delay problems. Care must be taken however, since traditional Runge–Kutta methods are not based directly on polynomial interpolants. With some of the available Runge–Kutta interpolants, it is necessary to consider separately the interpolation error and the integration error [1,13]. It will be seen that this is not necessary for the present methods. We will not consider the question of the relative merits of multistep and Runge–Kutta solvers in this paper but will concentrate on a particular Runge–Kutta delay solver.

There are several approaches to the question of interpolation for Runge–Kutta methods (for example see [7,8,39]). The present software uses continuously imbedded Runge–Kutta methods due to Sarafyan [32,34–38], who has investigated extensively the question of interpolation for Runge–Kutta methods. The first systematic use of interpolation for Runge–Kutta methods is described in [36]. (See also [34,35,37,38] and the references therein.) The present methods are based on $C^1$ piecewise polynomial interpolants. The underlying polynomial approximations are themselves Runge–Kutta approximations which are used directly in the error estimation and step size selection. Consequently, no problem arises due to the approximation polynomials not reflecting the local accuracy of the basic integration method.

The present interpolants also provide a natural means for handling root finding requirements. In addition to the type of root finding addressed in many standard ODE solvers, the interpolants may be used to handle automatically certain derivative discontinuities inherent in the solution of delay equations. The manner in which this attractive feature of the methods can be exploited in the solution of delay equations is described in the following sections. The mathematics on which the software is based is described in detail in [22,26–29] and [37,38]. Only summary details will be given in this paper.

The present software allows the inclusion of the delayed derivative in the differential equation. Thus, the software described can handle neutral equations. However, the user should
be aware of the characteristics of FDEs with SDLs discussed in Section 2 since the lack of smoothing for neutral equations may cause the software to fail. In practice, valid physical models, from which such equations are derived, are usually well-behaved and the software should perform well.

The remainder of this paper is divided in the following manner. In Section 2, the theoretical properties of FDEs and order of convergence results for fixed step methods are reviewed. In Section 3, the interpolation philosophy employed in conjunction with methods of Sarafyan is discussed. In Section 4, the basic software features and usage requirements are described. In Section 5, examples will be given which demonstrate the typical performance of the software. An expanded version of this paper may also be found in [30].

2. Theoretical properties of FDEs and order results

In order to simplify exposition, the scalar equations below will be used to illustrate the concepts of jump discontinuity propagation and numerical convergence results.

\[ \frac{dx(t)}{dt} = f(t, x(t), x(\alpha(x(t)))) \]  \hspace{1cm} (1)

or

\[ \frac{dx(t)}{dt} = f(t, x(t), x(\alpha(x(t))), x'(\alpha(x(t)))) \]  \hspace{1cm} (2)

for \( t \in [a, b] \) and with \( x(t) = \phi(t) \) for \( t \leq a \).

Throughout this discussion, it will be assumed that the functions \( f, \phi, \) and \( \alpha \) are continuously differentiable through derivatives of appropriate order. Precise derivative conditions can be found in the references cited. They will be omitted here since their repetition in this summary would be onerous. Equation (1) is a standard delay differential equation with state dependent lag, \( t - \alpha(x(t)) \), while equation (2) adds a derivative term on the right-hand side and is often called a “neutral” equation. Neutral equations have different error propagation properties, as well as different properties with respect to existence, uniqueness, and continuation of solutions. In this summary we shall concentrate on jump discontinuities and numerical accuracy properties critical to the numerical solution of such equations. While some of the results about neutral equations are new, they follow from extensions of proofs found in [11,25–27].

Derivative discontinuities

An initial discontinuity can occur when the initial function \( f \) is not “compatible” with the solution. This occurs whenever \( f \) does not satisfy the ordinary differential equation obtained by removing the delay over an initial interval by the “method of steps”, a technique for solving delay equations when lags are known to be positive, and only practical when delays are constant. (For a description of the method of steps, see [3].) It should not be surprising that initial functions are often not compatible. In the most general case, it is not necessary that one even demands continuity at the point \( t = a \). While we have ruled this out of equation (2), it only is required that one define \( \phi(a) = p \), for some value \( p \). For the rest of this discussion, it will be
assumed that \( \phi(t) \) is continuous at \( t = a \) from the left, and that consequently the solution is continuous at \( t = a \).

In general then, the first derivative of the solution \( x(t) \) can have a discontinuity at \( t = a \). Whenever the retarding function \( \alpha(t, x(t)) \) "crosses" \( t = a \), a potential for a first-order discontinuity to propagate exists. In fact, it was shown in [11] that discontinuities are zeroes \( Z \) with odd multiplicities of

\[
\alpha(t, x(t)) - Y = 0. 
\]

where \( Y \) is either \( t = a \), or any subsequent point of derivative discontinuity. (The "odd" multiplicity guarantees that the retarding function actually "crosses" the previous jump point.) Thus, all jump points "cascade" from the initial discontinuity at \( t = a \) through equation (3). Discontinuities can be assigned a unique precedence relationship. Every jump point has a unique "ancestor." Thus, \( Y \) is said to be the ancestor of the zero \( Z \) in equation (3).

**Primary jump point facts**

Generally, each propagated jump point can be a point where a derivative of \( x(t) \) of some order is discontinuous. Following is a summary of important facts and relevant results related to the question of locating such jump points.

**Generalized smoothing.** For equation (1), \( x(t) \) is discontinuous at the propagated jump point in, at worst, the next higher derivative than at the ancestor jump point.

**Theorem 1** (Jump order theorem). If the jump point \( Y \) is a point where the lowest-order derivative discontinuity occurs in the \((d - 1)\)st derivative, then \( x(t) \) will be continuous through, at least, its first \( z \cdot d \) derivatives, where \( z \) is the multiplicity of the zero \( Z \) of equation (3).

The significance of \( z \) will be made clear in the numerical results discussed below. If \( z \) is odd and greater than 1 it introduces greater smoothing in the propagated derivative discontinuity than if \( z = 1 \), the usual case.

**Neutral equations lack smoothing.** For equation (2) each propagated jump point can have a discontinuity in the same-order derivative as at the previous jump point. In fact, it is possible to have a cluster of jump points with derivative discontinuities in the first derivative, even if \( \alpha \) is monotone. This could occur at any fixed point of \( \alpha \), i.e., at a point \( T_f \) where the lag vanishes, \( \alpha(T_f, x(T_f)) = T_f \). See [30] for an example of such a problem.

**Order-h convergence results**

The importance of the above results is evidenced in order of convergence results for numerical methods. Clearly, low-order derivative discontinuities can render high-order numerical methods inaccurate. In software employing adaptive step size methods, a violation in continuity conditions can invalidate the step size selection heuristics and result in totally
inaccurate results. This is most disconcerting in equations of the form of equations (1), (2), in that the discontinuities can occur even if the problem is defined with \(C^1\) functions, \(f\), \(\phi\), and \(\varphi\).

For ordinary differential equations, it is well known that if the jump points are known and included in the discretization mesh, one-step methods or suitably modified multi-step methods can maintain order of convergence. This result extends to FDEs of the form of equation (2) provided interpolation of past points is not done “across” a low-order derivative discontinuity. The software described later pays careful attention to this fact. Since the jump points in the state-dependent lag case are not known a priori, they must be calculated. The issues become (1) how accurately can the jump points be calculated, and (2) whether the accuracy is sufficient to maintain the desired order of convergence for higher-order methods.

The underlying assumptions will be that all numerical calculations are being carried out by a one-step numerical differential equation method, that all calculated jump points are included in the basic mesh, and that the maximum step size taken in \(h\). Throughout we assume that \(Z_h\) is a root of

\[
\alpha(t, x_h(t)) - Y_h = 0, \tag{4}
\]

where \(x_h(t)\) is a \(p\)th-order approximation to \(x(t)\), and where \(Y_h\) is an approximation to the jump point \(Y\), the ancestor of \(Z\). Since \(Y\) and \(Z\) are propagated jump points, it will be assumed also they are zeroes of odd multiplicities \(y\) and \(z\) of their respective jump point equations (i.e., equations of the form of equation (3)). Further, for simplicity, we assume the liberal \(C^\infty\) conditions on \(f\), \(\phi\), and \(\alpha\). More restrictive and precise assumptions can be found in [11].

**Theorem 2.** If

\[
|Y - Y_h| = O(h^n), \tag{5}
\]

\[
|Z - Z_h| = O(h^r), \tag{6}
\]

then

\[
|x(t) - x_h(t)| = O(h^Q), \text{ where } Q = \min(p, r \cdot z, n \cdot y),
\]

and \(x_h(t)\) is the \(p\)th-order numerical solution to equation (1), as described above.

This theorem establishes that if the products, \(r \cdot z\) and \(n \cdot y\), equal or exceed \(p\), \(p\)th-order convergence will be preserved in crossing approximate jump points when using a suitably modified \(p\)th-order method. It therefore sets a standard for “how accurate” the approximate jump points must be to maintain the \(p\)th-order convergence of the underlying numerical method. One can recursively apply the result to each group of propagated jump points and inductively show that \(p\)th-order convergence will be maintained. (The inductive step would involve showing that \(n \cdot y \geq p\), provided the solution to equation (4) were sufficiently accurate.) It will be assumed that the number of jump points is arbitrary, yet finite. For theoretical purposes, it will be assumed also that the solutions to equations of the form of equation (4) are obtained by simple bisection to arbitrary accuracy. (This may be accomplished by monitoring sign changes in \(\alpha(t, x(t)) - Y_h\), and adjusting the step size until the zero, \(t = Z_h\) is included in the mesh, for each \(Y_h\) in the tree of previous approximate jump points. Actually, the software discussed in later sections provides more robust alternatives to this root finding process.)
Even if $Z_h$ is computed to infinite accuracy as a zero of equation (4), we still need to establish "how accurate" an approximation it will be to $Z$. It is in this regard that neutral equations and standard delay equations are different. The accuracy depends directly on the generalized smoothing property, which neutral equations lack. To be specific, we restate the following result from [11], again with appropriate continuity assumptions.

**Theorem 3.** If

$$|Y - Y_h| = O(h^n),$$

then

$$|Z - Z_h| = O(h^{\min(p,n)/z}),$$

where $z$ is the multiplicity in $Z$ in equation (3). (Note: $x_h(t)$ in equation (3) is the $p$th-order numerical solution to equation (1).)

At this juncture it is important to note that the three theorems above for standard delay equations with state-dependent lags are sharp. That is, there are examples which show that no greater order of convergence can generally be achieved (see [12,26]).

Examining Theorems 2 and 3 reveals that, for $p$th-order methods applied to equation (1), $p$th-order convergence will be preserved for nonneutral problems. In fact, for standard equation (1), the accuracy loss due to multiplicity of the root $Z$ is compensated exactly by the increased derivative smoothness due to Theorem 1.

**Neutral equations**

The above theorems apply to neutral equations with a slight, but from a numerical perspective devastating, change. The property of generalized smoothing, as mentioned, does not apply to neutral equations. For example, if $Z$ is a zero of multiplicity $z = 1$ of equation (3), the derivative does not smooth, i.e., the discontinuity occurs in the same derivative as it does at the ancestor point $Y$. In general, Theorem 1 becomes the following.

**Theorem 1' (Jump order theorem for neutral equations).** If the jump point $Y$ is a point where the lowest-order derivative discontinuity occurs in the $(d - 1)$st derivative, then $x(t)$ will be continuous through, at least, its first $(z \cdot d - 1)$ derivatives, where $z$ is the odd multiplicity of the zero $Z$ of equation (3).

As was observed in the inductive arguments for nonneutral equations, the smoothing was "exactly" what was required to maintain $p$th-order convergence. The loss of one degree of smoothing is sufficient to lose one order of convergence for each propagated jump point "level." In practice, with adaptive step size methods, this may not be a severe degradation, yet it indeed invalidates the basic theory of convergence on which methods compute step size and maintain accuracy. The software discussed in this work provides for neutral equations, but should be used advisedly when a large number of propagated jump points occurs.
3. Sarafyan methods

The basic idea used in the derivation of continuously imbedded Runge–Kutta methods is to obtain a traditional discrete Runge–Kutta method which satisfies the order equations of order, say, \( r \), and then to find a polynomial approximation that uses the same derivative approximations and which also satisfies the Runge–Kutta order equations of some order \( p \). For example, in [34] Sarafyan obtained a fifth-order Runge–Kutta method in which is imbedded a third-order polynomial approximation. See [42] for a discussion of software based on these methods. See [35] for improvements of the methods in [34]. Reference [32] contains similar sixth-order methods with imbedded fourth-order interpolants. It is desirable that \( p \) equal \( r \) or \( r - 1 \), and Sarafyan has developed methods with this remarkable property. Specific imbeddings of this type are discussed in [41]. The use of one of the sets of methods discussed in [41] for the solution of delay equations is described in [43]. These methods include a seven-stage fifth-order Runge–Kutta method in which two fourth-order polynomial approximations are imbedded. A drawback of these methods is that the underlying polynomial interpolant is not \( C^1 \), which for some problems with root finding requirements (e.g., neutral equations!) can be a serious drawback [39]. Sarafyan has developed several sets of similar methods which contain \( C^1 \) interpolants [38]. Interestingly, these methods require seven stages per step but effectively are six-stage methods since the final derivative evaluation for each step is also the first evaluation for the next step. The use of one of these more recent sets of methods will be considered in this paper. The use of similar higher-order pairs will be considered elsewhere.

A standard coefficient tableau is associated with the methods chosen. The tableau contains the coefficients \( a_i \) and \( b_{ij} \) that determine the calculation of the Runge–Kutta derivative approximations. If the integration step size is denoted by \( h \), these approximations are defined by:

\[
\begin{align*}
    k_0 &= hf(t_n, x_n), \\
    k_i &= hf\left(t_n + a_ih, x_n + \sum_{j=0}^{i-1} b_{ij}k_j\right), \quad \text{for } i > 0.
\end{align*}
\]

The coefficient tableau for the basic method under consideration is given as follows:

\[
\begin{array}{ccc|ccc|ccc|ccc}
    i & a_i & b_{ij} & j = 0, \ldots, i - 1 \\
    1 & 1/6 & 1/6 & & & & & & & & & & \\
    2 & 1/4 & 1/6 & 3/16 & & & & & & & & & & \\
    4 & 3/4 & 16 & 0 & 0 & 9 & & & & & & & & & & \\
    5 & 1 & -4/7 & 3/7 & 12/7 & -12/7 & 8/7 & & & & & & & & & & \\
\end{array}
\]

Define the following:

\[
\begin{align*}
    \Omega_1 &= k_0, \\
    \Omega_2 &= (-25k_0 + 48k_2 - 36k_3 + 16k_4 - 84k_5 + 81k_6)/6, \\
    \Omega_2 &= (70k_0 - 208k_2 + 228k_3 - 112k_4 + 490k_5 - 468k_6)/9,
\end{align*}
\]
\[ \Omega_4 = \frac{-40k_0 + 144k_2 - 192k_3 + 112k_4 - 399k_5 + 375k_6}{6}, \]
\[ \Omega_5 = \frac{8(4k_0 - 16k_2 + 24k_3 - 16k_4 + 49k_5 - 45k_6)}{15}, \]
\[ \omega_1 = k_0, \]
\[ \omega_2 = \frac{-127k_0 + 144k_2 + 36k_3 - 80k_4 + 27k_6}{42}, \]
\[ \omega_3 = \frac{2(5k_0 - 8k_2 - 2k_3 + 8k_4 - 3k_6)}{3}, \]
\[ \omega_4 = \frac{2(-13k_0 + 24k_2 + 6k_3 - 32k_4 + 15k_6)}{21}. \]

Let \( c = (t - t_n)/h. \) Now define the following methods:

\[ x_{4,7,1}(t_n + ch) = x_n + \sum_{i=1}^{5} \Omega_i c^i, \]
\[ x_{4,7,2}(t_n + ch) = x_n + \sum_{i=1}^{4} \omega_i c^i. \]

Each of the above methods is a six- or seven-stage fourth-order Runge–Kutta approximation for each value of \( c. \) For \( c = 1, \) \( x_{4,7,1} \) is a six-stage fifth-order Runge–Kutta C1 approximation which will be denoted by \( x_{5,6}; \) and \( x_{4,7,2} \) is a seven-stage fourth-order method which will be denoted by \( x_{4,7}. \) These methods are defined by

\[ x_{5,6} = x_n + \frac{(7k_0 + 32k_2 + 12k_3 + 32k_4 + 7k_5)}{90}, \]
\[ x_{4,7} = x_n + \frac{(3k_0 + 16k_2 + 4k_3 + 16k_4 + 3k_5)}{42}. \]

In the software discussed in the next section, \( x_{5,6} \) is used as the basic integration method, \( x_{4,7,1} \) is used for interpolation and root finding, and \( x_{5,6} - x_{4,7} \) is used for error estimation. The use of other methods from [38] will be considered in more detail elsewhere.

### 4. Basic features of the software and usage requirements

The basic features of the software and the usage requirements for the software are summarized in this section. Many of the ODE heuristics in the software are modeled after the corresponding ones in the well-known Runge–Kutta–Fehlberg solver DERKF discussed in [40] (and the references therein). These heuristics include limiting the magnitude of step size reductions or increases, testing for the impossibility to continue with a pure relative error test, testing if the step size is too small for machine precision, and using a mixed absolute–relative integration error tolerance. (Several other DERKF heuristics are also used throughout the code.)

The methods discussed in [43] possess the particularly attractive feature that the maximum magnitude of the difference of the interpolants for \( 0 \leq c \leq 1 \) is always attained for \( c = 1. \) Therefore, the difference of the interpolants at off-mesh points \( (0 \leq c < 1) \) is guaranteed to be no larger than the difference at integration mesh points \( (c = 1) \). The situation is slightly different with the present methods, as summarized in the following result.

**Theorem 4.** Using the previous notation, the maximum difference

\[ \max\{ |(x_{4,7,1} - x_{4,7,2})(t_n + ch)| : 0 \leq c \leq 1 \} \]

between the imbedded Runge–Kutta polynomials is attained for \( c = \frac{3}{4}. \)
The above result may be established in the following manner. An argument similar to that in [43] shows that for any value of \( c \),

\[
(x_{4,7,1} - x_{4,7,2})' = p(c) \sum_{i=0}^{6} \alpha_i k_i,
\]

where the \( \alpha_i \) do not depend on \( c \) and the polynomial \( p(c) \) is given by

\[
p(c) = c(c - \frac{2}{3})(c - \frac{1}{3})(c - 1).
\]

Since \( x_{4,7,1} = x_{4,7,2} \) for \( c = 0 \), the maximum difference between \( x_{4,7,1} \) and \( x_{4,7,2} \) is thus attained for a value of \( c \) in \( \left\{ \frac{2}{3}, \frac{1}{3}, 1 \right\} \). A comparison of the magnitudes of \( x_{4,7,1} - x_{4,7,2} \) for these values of \( c \) shows that the maximum difference always occurs for \( c = \frac{2}{3} \) and the magnitude for this value is approximately 3.4 times larger than the magnitude for \( c = 1 \). If this factor is taken into account when the integration error is estimated, the difference at off-mesh points can be controlled to any desired degree of accuracy. Therefore, it is not necessary to consider the polynomial approximation error separately when subsequent interpolations are performed.

Following each integration step, the error is estimated using the maximum difference over all components of the ODE for \( c = 1 \). This difference is given by

\[
(x_{5,6} - x_{4,7}) = \left(4k_0 - 16k_2 + 24k_3 - 16k_4 + 49k_5 - 45k_6\right)/630
\]

(modulo the above factor of 3.4). A mixed error test is used as in [40]. The estimated error in each component is given by

\[
\frac{|(x_{5,6} - x_{4,7})(t_n + h)|}{\epsilon_a + \epsilon_r |x_{5,6}(t_n + h)|}
\]

where \( \epsilon_a \) and \( \epsilon_r \) are the absolute and relative error tolerances for the test.

As discussed in Section 2, a difficulty which must be considered is the effect of derivative discontinuities in the solution. In addition to handling simple time-dependent discontinuities, the software is capable of handling state-dependent derivative discontinuities. Following is a brief discussion of how this is accomplished using Sarafyan methods. The original ODE solvers (on which the present delay software is based) which are discussed in [41] contain provisions to solve the following problem:

\[
dx(t)/dt = f(t, x(t)),
\]

\( x(t_0) = x_0, \)

\( 0 = g(t, x(t), x'(t)). \)

In addition to solving a system of ODEs, it is therefore possible to locate roots of auxiliary “event” functions \( g(t, x(t), x'(t)) \) that depend on the solution. (This is a feature of several well-known standard ODE solvers.) Basically, the interpolation polynomial \( x_{4,7,1} \) is used in conjunction with a root finder to do the necessary root finding when a sign change in the event function is detected. (Recall that we are only interested in zeroes of equation (3) which have odd multiplicity.) In [29], Neves and Feldstein characterize the points at which derivative discontinuities can occur as roots of odd multiplicity (at which sign changes must occur) of functions of the form of equation (3) of Section 2.
The software discussed in [41] was modified to automatically locate roots of these functions. (In addition, the software allows the user to supply other root functions and to conveniently make necessary changes in problem parameters at roots of the functions.) Note that this entails working with a system of root functions which increases in size each time a root is located. In general, the roots form a tree of points [29]. The basic idea is to have the software automatically locate each discontinuity point in the tree. Extensive testing indicates that the modified software is capable of doing this. Location of the roots makes it possible to include each as an integration mesh point, thus avoiding the difficulty of interpolating across discontinuities. The present software attempts to locate all such points of discontinuity. No attempt is made to determine if the discontinuity is numerically significant (that is, if it occurs in a high-order derivative which does not affect the accuracy of the integration) or to eliminate such points from the root finding. Since the user is allowed to specify additional root functions, it is possible for the user to employ techniques such as those discussed in [27], if it is desirable to do so.

For problems that do not involve derivative discontinuities and problems in which the derivative discontinuities can be ignored (this is often the case for nonneutral problems as indicated in Theorem 1 due to generalized smoothing), the automatic root finding (which is a user-controllable option) need not be used. The ability to control the use of root finding is particularly important since experience indicates that discontinuities can simply be ignored for many problems of interest.

A circular history queue is used to save the necessary Runge–Kutta derivative approximations following each integration step. Before each derivative evaluation is requested, the queue is searched and the necessary interpolations for the delayed solution variables are performed. The software also contains modules which allow the user to interpolate the solution and/or the derivative at times other than the delay times. This is often useful for plotting purposes.

If the lag vanishes at isolated points, a problem is said to be a “vanishing lag problem”. Vanishing lag problems pose serious challenges for delay software. The present software contains two solution options. With the first option (which is appropriate for nonvanishing lag problems), once the integration is started, the step size is limited to prevent a delay time from falling beyond the history queue (by bisecting the integration step, if necessary, until the delay falls within an integration interval spanned by the history queue). The intent of this restriction is to avoid extrapolation of the previous polynomial interpolant beyond the last successful integration time in approximating the delayed variables. With this option, the software is not capable generally of solving vanishing lag problems due to the step size becoming too small near the points at which the lag vanishes (due to the integration step bisections). The second option (which is appropriate for vanishing lag problems) is somewhat different. It is desirable to avoid extrapolating the previous solution polynomial because the extrapolated values may not be accurate and because the code might extrapolate across a discontinuity. With this option, at each step the code chooses a step size $h$ for which the solution is actually accurate for a step size of $2h$. This amounts to using $c = 2$ in the error test rather than $c = 1$. If a delay falls beyond the integration history queue on the next step (and the step size has been increased), the step size is reset to its previous value and extrapolation is used to approximate the delayed variables. The extrapolation is accurate due to the manner in which the step size $h$ was chosen.

Another approach would be to incorporate special techniques (such as those in [25,26]) to handle vanishing lags. There is one special situation in which these special techniques are used...
in the present software. The extrapolatory technique described above requires that an integration step be completed successfully before it is applicable. For some problems (e.g., \( x'(t) = x(t/2) \) for \( t \geq 0 \)), the difficulty arises on the very first integration step. The technique is not directly applicable for such problems. The self-starting methods of Tavernini discussed in \([25, 26]\) are used in this case to construct an interpolant for the delayed variables. This provides an approximation which has the same order of convergence as the Sarafyan methods for the first step. In all other situations, the entire solution is based on the Sarafyan methods.

Nondelay equations pose no difficulty for the software, that is, the software is capable of solving problems in which the lag for one or more solution components vanishes identically (i.e., components which do not contain delays). For example, the software has been used to solve successfully all of the nondelay problems in the well-known nonstiff ODE test set \([9]\) of Enright et al. The tests actually demonstrate that the software is competitive with other well-known ODE solvers when used to solve nondelay equations \([41]\). It is remarked that there is a need for the development of a systematic test environment for delay equations similar to the ODE test environment described in \([9]\).

The documentation prologues and programming conventions used in the software adhere to the standards adopted for the well-known SLATEC \([5]\) mathematical software library. Machine dependencies and error handling are also handled in a manner consistent with the SLATEC library in order to increase portability. The software attempts to diagnose several nonstandard errors peculiar to the solution of delay equations. The software is written in FORTRAN since it is intended for eventual inclusion in the SLATEC library.

As is customary with some other software for delay equations, the delay or retarding function is not allowed to change from one equation in the system to another, that is, if \( \alpha_k(x_k(t), t) \) appears in the derivative for the \( k \)th solution component, it is not permissible for a term \( \alpha_j(x_j(t), t) \) to appear in the derivative for another solution component unless \( \alpha_j = \alpha_k \). This is not a real restriction since it can be avoided using the process of "augmenting the system." \([26]\) contains a detailed discussion of this issue. See \([4]\) for an example of a problem of physical interest that can be solved using the present software and this process.

The software allows the user to provide the initial step size. (This is convenient for some problems with rootfinding requirements.) If the user chooses not to do this, the software uses a modification of a well-known initial step size selection subroutine HSTART \([44]\) to calculate the initial step size. The modification uses the initial delay function and, if necessary, the methods of Tavernini to approximate the delayed solution when HSTART requires a derivative evaluation.

The software contains several internal hooks which are transparent to the user. It is possible to use the software in a fixed step size manner. While this is not generally a good idea, such usage is sometimes required to incorporate the software into existing control programs. It is possible to use an error per unit step criterion rather than the customary error per step criterion. It is possible to alter the amount of accuracy requested in the root finding. Currently, the software performs the root finding as accurately as possible regardless of the integration error tolerance. (In light of the propagation theorems discussed above, this seems particularly important, especially for neutral equations.) The software does not allow extrapolation of the oldest Runge-Kutta polynomial in the event a delay time falls before the oldest point in the history queue. (The size of the circular history queue is determined by the user. Consequently, it is necessary to discard the oldest integration information when the history queue is full.)
Similarly, it does not allow extrapolation of the most recent polynomial in the event a delay time falls beyond the most recent point in the history queue (unless the nonvanishing lag option described above is used). For some but certainly not all problems, allowing one or both of these extrapolations still yields satisfactory results without a degradation in accuracy. The present software allows the user to reduce the step size and effectively restart the integration each time the root of an event function is located (if the user does not reduce the step size, the code continues with the step size last used successfully before the root was located). It is possible to have the software simply continue the integration from the last successful mesh time rather than from the root. It is possible to bypass the extrapolatory root calculations. The only reason they are performed is to limit the step size in order to minimize the chance of error test failures for problems with discontinuities. With a loss in efficiency due to more rejected steps, their use can be eliminated. It is possible to use \( x_{4.7} \) as the basic integration method and \( x_{4.7.2} \) for root finding and interpolation. The present software uses \( x_{5.6} \) and \( x_{4.7.1} \) for these tasks. The integration error test is based on the magnitude of \( x_{5.6} - x_{4.7} \). Although this is a fourth-order approximation, it is treated as a fifth-order approximation since \( x_{5.6} \) is fifth-order. This amounts to using local extrapolation [40]. However, at off-mesh points, the difference \( x_{4.7.1} - x_{4.7.2} \) is strictly a fourth-order approximation since \( x_{4.7.1} \) and \( x_{4.7.2} \) are each fourth-order. If a strictly consistent fourth-order code is desired, the switch to \( x_{4.7} \) and \( x_{4.7.2} \) can be made.

Using any of the above hooks may be accomplished by changing one corresponding statement in the code. Experience indicates that the software generally performs more efficiently and more reliably for a given accuracy using the present options. However, sophisticated users, software developers, and skeptics may find one or more of these hooks useful for particular problems.

The software is organized in levels. A third-level driver resembles the second-level step-oriented drivers contained in most standard ODE solvers. This driver advances the integration one step and performs the necessary tasks for continuing the integration. A second-level driver resembles the first-level interval-oriented drivers contained in most standard ODE solvers. This driver integrates the problem over a specified interval. Although the software can be used via these drivers, a nonstandard first-level driver DRKLAG is the one which it is anticipated most users will use. DRKLAG may be described as a "problem-oriented" driver.

To use the DRKLAG driver, the user must define the number of equations and provide the necessary workspace for the problem. In addition, the user must provide the EXTERNAL names of several subroutines in which various tasks are performed. SUBROUTINE INITIAL is required to initialize the problem (e.g., define such parameters as the beginning and final integration times, the local error tolerances, the initial step size, and solver options). SUBROUTINE DERIVS is required to calculate the system derivatives given approximations to the current solution and the delayed variables. SUBROUTINE GUSER is required to evaluate the residuals for the user-defined root functions given approximations to the current solution and derivative. SUBROUTINE BETA is required to evaluate the system lags (denoted by \( \alpha - t \) in this paper) given an approximation to the current solution. SUBROUTINE YINIT is required to evaluate the solution for delay times prior to the initial integration time. Similarly, SUBROUTINE DYINIT is required to evaluate the derivative for delay times prior to the initial integration time for neutral problems. SUBROUTINE DPRINT is required to process the solution at output points (in a manner similar to that employed in some of the solvers from the well-known NAG and HARWELL libraries). Finally, SUBROUTINE CHANGE is needed
for problems with root finding requirements. This subroutine is called each time a zero of one of the root functions is located to permit the user to make changes in problem parameters and integration options. DRKLAG does not return control to the calling program until the final integration time is reached. [30] contains a detailed description of the necessary subroutines for the solution of a typical problem.

Thus, most communication with the DRKLAG driver is via user-provided subroutines. While this is different than the user interface for most standard ODE solvers, we have found that this approach helps the user break the problem into manageable pieces. This seems to be particularly useful for a software package such as the present one which requires more complicated interaction with the user than does a standard ODE solver.

5. Numerical examples

In this section examples are used to illustrate the manner in which delay problems can be solved using the present software. All results reported in this section were obtained using equal absolute and relative error tolerances, $\varepsilon = \varepsilon_a = \varepsilon_r$, for a given values of $\varepsilon$. An abbreviated notation is used in the tables; e.g., 348-3 means 0.348E-3. All results were obtained using the double precision version of the software on an IBM PC compatible computer. The examples given in this section are intended to illustrate the performance of the software for typical problems. [30] contains the results for several additional problems including neutral equations and vanishing lag problems.

Example 1 [13,19,43].

$$\frac{dx(t)}{dt} = x(t-1), \quad \text{for } 1 \leq t \leq 15,$$

$$x(t) = 1, \quad \text{for } -1 \leq t \leq 0.$$  

For each integer $k \geq 0$, the exact solution for this constant-lag problem is a polynomial of degree $k$ on the interval $[k-1, k]$ and is given for $k = 0, 1, \ldots$ by:

$$x(t) = \sum_{i=0}^{k} \frac{(t-i+1)^{i}}{i!}, \quad \text{for } t \in [k-1, k].$$

Jump discontinuities in the $(k+1)$st derivative occur at $t = k$, $k = 1, \ldots$. This illustrates the smoothing property discussed earlier. Table 1 contains the maximum global relative errors for

<table>
<thead>
<tr>
<th>$-\log_{10} \varepsilon$</th>
<th>Number of derivative evaluations</th>
<th>Maximum global relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>196</td>
<td>104-7</td>
</tr>
<tr>
<td>8</td>
<td>394</td>
<td>438-9</td>
</tr>
<tr>
<td>10</td>
<td>658</td>
<td>152-11</td>
</tr>
<tr>
<td>12</td>
<td>1456</td>
<td>882-14</td>
</tr>
</tbody>
</table>

Table 2

<table>
<thead>
<tr>
<th>$-\log_{10} \varepsilon$</th>
<th>Number of derivative evaluations</th>
<th>Maximum global relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>321</td>
<td>871-3</td>
</tr>
<tr>
<td>8</td>
<td>575</td>
<td>424-7</td>
</tr>
<tr>
<td>10</td>
<td>913</td>
<td>349-9</td>
</tr>
<tr>
<td>12</td>
<td>1887</td>
<td>137-10</td>
</tr>
</tbody>
</table>
Table 3
Results for Example 2, discontinuities located

<table>
<thead>
<tr>
<th>$-\log_{10} \varepsilon$</th>
<th>Number of derivative evaluations</th>
<th>Maximum global relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>275</td>
<td>208-2</td>
</tr>
<tr>
<td>5</td>
<td>419</td>
<td>623-5</td>
</tr>
<tr>
<td>7</td>
<td>551</td>
<td>105-5</td>
</tr>
<tr>
<td>9</td>
<td>1109</td>
<td>441-6</td>
</tr>
</tbody>
</table>

Table 4
Results for Example 2, discontinuities ignored

<table>
<thead>
<tr>
<th>$-\log_{10} \varepsilon$</th>
<th>Number of derivative evaluations</th>
<th>Maximum global relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>150</td>
<td>943-2</td>
</tr>
<tr>
<td>5</td>
<td>336</td>
<td>521-3</td>
</tr>
<tr>
<td>7</td>
<td>792</td>
<td>207-5</td>
</tr>
<tr>
<td>9</td>
<td>1656</td>
<td>432-6</td>
</tr>
</tbody>
</table>

$t \leq 15$ and the numbers of derivative evaluations required to solve this problem for various error tolerances where the automatic root finding option was used to locate the discontinuities. Table 2 contains the corresponding statistics where the discontinuities were ignored. For each tolerance, use of the root finding is both more efficient and more accurate.

**Example 2** [26,43].

\[
\frac{dx(t)}{dt} = (1/t)x(t)x(\ln(x(t))), \quad \text{for } t \geq 1,
\]

\[
x(t) = 1, \quad \text{for } 0 \leq t < 1.
\]

The exact solution for this problem for $t \leq e_3$ is given by:

\[
x(t) = \begin{cases} 
  t, & \text{if } 1 \leq t \leq e, \\
  \exp(t/e), & \text{if } e \leq t \leq e^2, \\
  \left(\frac{e}{3 - \ln(t)}\right)^e, & \text{if } e^2 \leq t \leq e_3,
\end{cases}
\]

where $e_3 = \exp(3 - \exp(1 - e))$.

For this problem, derivative jump discontinuities occur at $t = e$, $t = e^2$, and $t = e_3$. These points cannot be located without a knowledge of the solution since the delay functions are state-dependent. Table 3 contains the maximum global relative errors for $t \leq 17$ and the numbers of derivative evaluations required to solve this problem for various error tolerances where the automatic root finding option was used to locate the discontinuities. Table 4 contains the corresponding statistics where the discontinuities were ignored.

**Example 3** [18].

\[
\frac{dx_1(t)}{dt} = x_1'(t-1) + 4x_2(t), \quad \text{for } 0 \leq t \leq 2,
\]

\[
\frac{dx_2(t)}{dt} = x_1(t) - x_1(t-1),
\]

\[
x_1(t) = \exp(-2t),
\]

\[
x_2(t) = \frac{1}{2}(\exp(-2(t-1)) - \exp(-2t)), \quad \text{for } t \leq 0.
\]

This neutral problem is equivalent to the following integral equation solved in [18]:

\[
\frac{dx_1(t)}{dt} = x_1'(t-1) + 4 \int_{t-1}^{t} x_1(s) \, ds,
\]

\[
x_1(t) = \exp(-2t), \quad \text{for } t \leq 0.
\]
Table 5
Results for Example 3

<table>
<thead>
<tr>
<th>$-\log_{10}\varepsilon$</th>
<th>Number of derivative evaluations</th>
<th>Maximum global relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>128</td>
<td>129-0</td>
</tr>
<tr>
<td>4</td>
<td>146</td>
<td>292-1</td>
</tr>
<tr>
<td>5</td>
<td>164</td>
<td>576-2</td>
</tr>
<tr>
<td>6</td>
<td>224</td>
<td>317-3</td>
</tr>
<tr>
<td>7</td>
<td>296</td>
<td>113-3</td>
</tr>
<tr>
<td>8</td>
<td>422</td>
<td>138-4</td>
</tr>
<tr>
<td>9</td>
<td>626</td>
<td>332-5</td>
</tr>
<tr>
<td>10</td>
<td>974</td>
<td>552-6</td>
</tr>
<tr>
<td>11</td>
<td>1418</td>
<td>994-7</td>
</tr>
<tr>
<td>12</td>
<td>2402</td>
<td>167-7</td>
</tr>
<tr>
<td>13</td>
<td>3770</td>
<td>291-8</td>
</tr>
<tr>
<td>14</td>
<td>5756</td>
<td>893-9</td>
</tr>
<tr>
<td>15</td>
<td>9308</td>
<td>610-10</td>
</tr>
</tbody>
</table>

The exact solution for the problem is given by:

$$x_1(t) = \exp(-2t),$$

$$x_2(t) = \frac{1}{2}(\exp(-2(t-1)) - \exp(-2t)).$$

As a matter of interest, it is remarked that the manner in which this problem is solved demonstrates the applicability of the present software to the solution of Volterra integral equations. This problem was included as an example of a neutral problem which can be solved using DRKLAG. Since the solution for this problem is smooth, the automatic root finding option was not used. Table 5 contains the maximum global relative errors and the numbers of derivative evaluations required to solve this problem for various error tolerances. Note that the error is reduced as the error tolerance is reduced.

6. Summary

This paper endeavored to present a robust approach to offering general purpose software for solving a broad class of delay differential equations. Continuously imbedded methods of Sarafyan were implemented in a robust ODE solver. The peculiar features of delay equations were summarized and appropriate modifications to the original ODE solver were made. This was accomplished while maintaining the error control and style familiar to ODE solvers. Moreover, the software is still efficient and effective on ODE problems through the user inputs. The storage of history data, the calculation of points of discontinuity, and rigorous treatment of the often anomalous behavior of delay equations were addressed. Several test problems and results were presented that demonstrate both theoretical and computational aspects of state-dependent delay equations. The software, which was developed for the well-known public domain SLATEC mathematical software library [5], is available in both single and double precision versions from the authors.
Acknowledgement

The research of the second author was performed in Oak Ridge, Tennessee at the Mathematical Sciences Section of the Oak Ridge National Laboratory under the auspices of the Faculty Research Participation Program of Oak Ridge Associated Universities, and supported by the Applied Mathematics Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC05-84OR21400 with Martin-Marietta Energy Systems, Inc. The authors are indebted to Diran Sarafyan for his kind permission to implement and use his recent, as yet unpublished, methods in this paper and in the software.

References


