Applied Numerical Mathematics 37 (2001) 441–458

APPLIED NUMERICAL MATHEMATICS

www.elsevier.nl/locate/apnum

Solving DDEs in MATLAB

L.F. Shampine [a,*], S. Thompson [b]

[a] *Mathematics Department, Southern Methodist University, Dallas, TX 75275, USA*

[b] *Department of Mathematics & Statistics, Radford University, Radford, VA 24142, USA*

Abstract

We have written a program, `dde23`, to solve delay differential equations (DDEs) with constant delays in MATLAB. In this paper we discuss some of its features, including discontinuity tracking, iteration for short delays, and event location. We also develop some theoretical results that underlie the solver, including convergence, error estimation, and the effects of short delays on stability. Some examples illustrate the use of `dde23` and show it to be a capable DDE solver that is exceptionally easy to use. © 2001 IMACS. Published by Elsevier Science B.V. All rights reserved.

1. Introduction

Our goal is to make it as easy as possible to solve effectively a large class of delay differential equations (DDEs). We restrict ourselves to systems of equations of the form

$$y'(x) = f\bigl(x, y(x), y(x-\tau_1), y(x-\tau_2), \ldots, y(x-\tau_k)\bigr) \tag{1}$$

for constant delays τ_j such that $\tau = \min(\tau_1, \ldots, \tau_k) > 0$. The equations are to hold on $a \leqslant x \leqslant b$, which requires the history $y(x) = S(x)$ to be given for $x \leqslant a$. Although DDEs with delays (lags) of more general form are important, this is a large and useful class of DDEs. Indeed, Baker et al. [1] write that "The lag functions that arise most frequently in the modeling literature are constants." Restricting ourselves to this class of DDEs makes possible a simpler user interface and more robust numerical solution.

A popular approach to solving DDEs exploits the fact that the task has much in common with the solution of ordinary differential equations (ODEs). Indeed, the DDE solver `dde23` that we discuss here is closely related to the ODE solver `ode23` from the MATLAB ODE Suite [18]. In particular, our attention is focused on the Runge–Kutta triple BS(2,3) used in `ode23` because it has some special properties that we exploit in `dde23`. (Refer to the survey by Zennaro [21] for a delightful discussion of the important issues for more general methods of this type.) In Section 2 we explain how explicit Runge–Kutta triples can be extended and used to solve DDEs. This extension is greatly simplified if the

[*] Corresponding author.
E-mail addresses: lshampin@mail.smu.edu (L.F. Shampine), thompson@runet.edu (S. Thompson).

maximum step size H is smaller than τ because the resulting formulas are explicit. This is no restriction when proving convergence as $H \to 0$, so many papers assume it. However, we believe that it is not acceptable in a quality solver, so we also discuss how to evaluate the implicit formulas that arise when the step size is bigger than τ. In Section 3 we prove uniform convergence of the numerical scheme in realistic circumstances. In the following section we justify an estimate of the local truncation error.

Generally $y'(a+)$ from (1) is different from $y'(a-) = S'(a-)$. It is characteristic of DDEs that this discontinuity in y' propagates as a discontinuity in y'' at $a + \tau_1, \ldots, a + \tau_k$. In turn these discontinuities propagate as a discontinuity in $y^{(3)}$, and so forth. Locating and accounting for low-order discontinuities in the solution is key to the use of Runge–Kutta methods for solving DDEs. In Section 5 we discuss the practical issues of locating these discontinuities. Any DDE solver must account for them, but `dde23` also deals with discontinuous changes in f at known points x in both the history and the interval of integration in a similar way.

Some problems involve changes in f at points that depend on the solution itself, or more generally on a function of the solution called an event function. Event functions are used to locate where y satisfies certain conditions, e.g., where a specific component of $y(x)$ has a local maximum. `dde23` has a powerful event location capability that is discussed in Section 6. It is difficult to deal with the many possibilities that are seen in practice, so providing this capability affected profoundly the design of the solver.

The natural step size of a Runge–Kutta method is the largest that will yield the desired accuracy. Delays that are long compared to this step size and delays that are short present special difficulties for a DDE solver that we consider in Section 7. In this we prove a result about the stability of the numerical scheme for small delays.

We close with a pair of examples that show restricting the class of problems, algorithmic developments, language facilities in MATLAB, and design have resulted in a DDE solver that is both capable and exceptionally easy to use. We have prepared a tutorial that shows how to use `dde23`. The solver and its auxiliary functions, the tutorial, and complete solutions of many examples from the literature are available at http://www.runet.edu/~thompson/webddes/.

2. Formulas

Explicit Runge–Kutta triples are a standard way to solve the ODE problem $y' = f(x, y)$ on $[a, b]$ with given $y(a)$. They can be extended to solve DDEs. Indeed, `dde23` is closely related to the ODE solver `ode23` [18] which implements the BS(2,3) triple [2]. A triple of s stages involves three formulas. Suppose that we have an approximation y_n to $y(x)$ at x_n and wish to compute an approximation at $x_{n+1} = x_n + h_n$. For $i = 1, \ldots, s$, the stages $f_{ni} = f(x_{ni}, y_{ni})$ are defined in terms of $x_{ni} = x_n + c_i h_n$ and

$$y_{ni} = y_n + h_n \sum_{j=1}^{i-1} a_{ij} f_{nj}.$$

The approximation used to advance the integration is

$$y_{n+1} = y_n + h_n \sum_{i=1}^{s} b_i f_{ni}.$$

For brevity we may write this in terms of the increment function

$$\Phi(x_n, y_n) = \sum_{i=1}^{s} b_i f_{ni}.$$

The solution satisfies this formula with a residual called the local truncation error, *lte*$_n$,

$$y(x_{n+1}) = y(x_n) + h_n \Phi\bigl(x_n, y(x_n)\bigr) + lte_n.$$

For sufficiently smooth f and $y(x)$ this error is $\mathrm{O}(h_n^{p+1})$. The triple includes another formula,

$$y^*_{n+1} = y_n + h_n \sum_{i=1}^{s} b_i^* f_{ni} = y_n + h_n \Phi^*(x_n, y_n).$$

The solution satisfies this formula with a local truncation error lte_n^* that is $\mathrm{O}(h_n^p)$. This second formula is used only for selecting the step size. The third formula has the form

$$y_{n+\sigma} = y_n + h_n \sum_{i=1}^{s} b_i(\sigma) f_{ni}.$$

The coefficients $b_i(\sigma)$ are polynomials in σ, so this represents a polynomial approximation to $y(x_n + \sigma h_n)$ for $0 \leqslant \sigma \leqslant 1$. We assume that this formula yields the value y_n when $\sigma = 0$ and y_{n+1} when $\sigma = 1$. For this reason the third formula is described as a continuous extension of the first. A much more serious assumption is that the order of the continuous extension is the same as that of the first formula. These assumptions hold for the BS(2,3) triple. For such triples we regard the formula used to advance the integration as just the special case $\sigma = 1$ of the continuous extension and write

$$y_{n+\sigma} = y_n + h_n \Phi(x_n, y_n, \sigma).$$

The local truncation error of the continuous extension is defined by

$$y(x_n + \sigma h_n) = y(x_n) + h_n \Phi\bigl(x_n, y(x_n), \sigma\bigr) + lte_n(\sigma).$$

We assume that for smooth f and $y(x)$, there is a constant C_1 such that $\|lte_n(\sigma)\| \leqslant C_1 h_n^{p+1}$ for $0 \leqslant \sigma \leqslant 1$.

In his discussion of continuous extensions, Dormand [4] treats the local truncation error of the BS(2,3) triple as an example. In Section 6.3 he obtains

$$A^{(4)}(\sigma) = \frac{\sigma^2}{288}\bigl(1728\sigma^4 - 7536\sigma^3 + 13132\sigma^2 - 11148\sigma + 3969\bigr)^{1/2} \tag{2}$$

as a measure of the local truncation error at $x_n + \sigma h_n$. In Fig. 6.2 he shows that this measure increases monotonely from zero at x_n to a maximum at x_{n+1}. We shall see that the solver controls the size of the error at x_{n+1}, so the continuous extension of this triple provides an accurate solution throughout $[x_n, x_{n+1}]$.

To use an explicit Runge–Kutta triple to solve the DDE (1), we need a strategy for handling the history terms $y(x_{ni} - \tau_j)$ that appear in

$$f_{ni} = f\bigl(x_{ni}, y_{ni}, y(x_{ni} - \tau_1), \ldots, y(x_{ni} - \tau_k)\bigr).$$

Two situations must be distinguished: $h_n \leqslant \tau$ and $h_n > \tau_j$ for some j. We begin with the first. Suppose that we have available an approximation $S(x)$ to $y(x)$ for all $x \leqslant x_n$. If $h_n \leqslant \tau$, then all the $x_{ni} - \tau_j \leqslant x_n$ and

$$f_{ni} = f\bigl(x_{ni}, y_{ni}, S(x_{ni} - \tau_1), \ldots, S(x_{ni} - \tau_k)\bigr)$$

is an explicit recipe for the stage and the formulas are explicit. The function $S(x)$ is the initial history for $x \leqslant a$. After taking the step to x_{n+1}, we use the continuous extension to define $S(x)$ on $[x_n, x_{n+1}]$ as $S(x_n + \sigma h_n) = y_{n+\sigma}$. We are then ready to take another step. This suffices for proving convergence as the maximum step size tends to zero, but we must take up the other situation because in practice, we may very well want to use a step size larger than the smallest delay. When $h_n > \tau_j$ for some j, the "history" term $S(x)$ is evaluated in the span of the current step and the formulas are defined implicitly. In this situation we evaluate the formulas with simple iteration. On reaching x_n, we have defined $S(x)$ for $x \leqslant x_n$. We extend its definition somehow to $(x_n, x_n + h_n]$ and call the resulting function $S^{(0)}(x)$. A typical stage of simple iteration begins with the approximate solution $S^{(m)}(x)$. The next iterate is computed with the explicit formula

$$S^{(m+1)}(x_n + \sigma h_n) = y_n + h_n \Phi\bigl(x_n, y_n, \sigma; S^{(m)}(x)\bigr).$$

Here and elsewhere it is convenient to show the dependence on the history by means of another argument in the increment function. In the next section we show that if the step size is small enough, this is a contraction and $S(x)$ is well-defined for $x \leqslant x_{n+1}$.

In `dde23` we predict $S^{(0)}(x)$ to be the constant y_0 for the first step. We do not attempt to predict more accurately then because the solution is not smooth at a and we do not yet know a step size appropriate to the scale of the problem. Indeed, reducing the step size as needed to obtain convergence of simple iteration is a valuable tactic for finding an initial step size that is on scale, cf. [16]. After the first step, we use the continuous extension of the preceding step as $S^{(0)}(x)$ for the current step. This prediction has an appropriate order of accuracy and we can even assess the accuracy quantitatively using (2). Remarkably, the truncation error has a local maximum at x_{n+1} so that extrapolation actually provides a more accurate solution for some distance. Specifically, the ratio of the error of $y_{n+\sigma}$ to the error of y_{n+1} is no larger than 1 for σ up to about 1.32. Much more important in practice is the accuracy of the prediction in the span of a subsequent step of the same size or rather larger. Unfortunately, the ratio grows rapidly as σ increases. Still, it is no bigger than 13 for $1 \leqslant \sigma \leqslant 2$, so the prediction is quite good when the step size is changing slowly. The step size adjustment algorithms of `dde23` are those of `ode23` augmented to deal with implicit formulas. The convergence test for simple iteration is that $\|y_{n+1}^{(m+1)} - y_{n+1}^{(m)}\|$ is no more than one tenth the accuracy required of y_{n+1}. Because each iteration costs as much as an explicit step, any proposed h_n with $\tau < h_n < 2\tau$ is reduced to τ so as to make the formulas explicit. Considering the quality of the prediction when $h_n \geqslant 2\tau$, we have allowed up to 5 iterations. If convergence is not achieved, h_n is halved and the step repeated. The solver cannot crash because of repeated convergence failures because eventually it will resort to a step size for which the formulas are explicit.

3. Convergence

Theorem. *Suppose an explicit Runge–Kutta triple is used to solve the DDE* (1) *as described in the preceding section. Assume that the meshes* $\{x_n\}$ *include all discontinuities of low orders and that the maximum step size* H *satisfies* (5) *and* (7). *If* f *satisfies a Lipschitz condition in its dependent variables and is sufficiently smooth in all its variables, then there exists a constant* C *such that for* $a \leqslant x \leqslant b$,

$$\bigl\|y(x) - S(x)\bigr\| \leqslant CH^p. \tag{3}$$

This theorem shows how our method works. A more general convergence result may be found in [21]. Before proving this result, we begin our investigation of convergence by recalling the comments earlier about how discontinuities arise and propagate for DDEs. We know in advance all the points where the solution might not be smooth enough for the usual orders of the formulas to hold. Because we consider one-step methods, if we take these points to be mesh points, the analysis is the same as if the solution were smooth throughout the interval. Accordingly, we consider a sequence of meshes $\{x_n\}$ which include all these points. Just how this is accomplished in practice is described in Section 5. Between these points we assume that f and $y(x)$ are smooth enough for the formulas to have their usual orders. We also assume that f satisfies a Lipschitz condition with respect to all the dependent variables. For the sake of simplicity, we suppose that there is only one delay and take the Lipschitz condition in the form

$$\left\| f(x,\tilde{y},\tilde{z}) - f(x,y,z) \right\| \leqslant L \max\bigl(\|\tilde{y}-y\|, \|\tilde{z}-z\|\bigr).$$

The Runge–Kutta formulas extended to DDEs involve a history term that we write generically in the increment function as $g(x)$. We require two lemmas about how the increment functions depend on their arguments.

Lemma 1. *There is a constant $\mathcal{L}$ such that for $0 \leqslant \sigma \leqslant 1$,*

$$\left\| \Phi\bigl(x_n,\tilde{y}_n,\sigma; g(x)\bigr) - \Phi\bigl(x_n,y_n,\sigma;g(x)\bigr) \right\| \leqslant \mathcal{L}\|\tilde{y}_n - y_n\|. \tag{4}$$

Proof. In a step of size h_n from $(x_n,\tilde{y}_n)$, the intermediate quantities $\tilde{y}_{ni}$ are defined by

$$\tilde{y}_{ni} = \tilde{y}_n + h_n \sum_{j=1}^{i-1} a_{ij} f\bigl(x_{nj},\tilde{y}_{nj}, g(x_{nj}-\tau)\bigr)$$

and the y_{ni} are defined similarly in a step from (x_n,y_n). Using the Lipschitz condition on f, it is straightforward to show that for each i, there is a constant L_i such that

$$\|\tilde{y}_{ni} - y_{ni}\| \leqslant \left(1 + HL\sum_{j=1}^{i-1} L_j|a_{ij}|\right)\|\tilde{y}_n - y_n\| = L_i\|\tilde{y}_n - y_n\|.$$

With this result it follows easily that (4) holds with $\mathcal{L} = L\sum_{i=1}^{s} L_i \max(|b_i(\sigma)|)$. □

The second lemma we need is a bound on the effect of using different history terms.

Lemma 2. *Let Δ be a bound on $\|G(x) - g(x)\|$ for all $x \leqslant x_{n+1}$. If the maximum step size H is small enough that*

$$HL\max_i\left(\sum_{j=1}^{i-1}|a_{ij}|\right) \leqslant 1 \tag{5}$$

then there is a constant Γ such that

$$\left\| \Phi\bigl(x_n,y_n,\sigma;G(x)\bigr) - \Phi\bigl(x_n,y_n,\sigma;g(x)\bigr) \right\| \leqslant \Gamma\Delta. \tag{6}$$

Proof. Let the intermediate quantities of the two formulas be

$$r_{ni} = y_n + h_n\sum_{j=1}^{i-1} a_{ij} f\bigl(x_{nj}, r_{nj}, G(x_{nj}-\tau)\bigr)$$

and

$$s_{ni} = y_n + h_n \sum_{j=1}^{i-1} a_{ij} f\bigl(x_{nj}, s_{nj}, g(x_{nj} - \tau)\bigr).$$

The Lipschitz condition on f, inequality (5), and a simple induction argument show that for each i,

$$\|r_{ni} - s_{ni}\| \leqslant h_n \sum_{j=1}^{i-1} |a_{ij}| L \Delta \leqslant \Delta.$$

With this result it follows easily that (6) holds with $\Gamma = L \sum_{i=1}^{s} \max(|b_i(\sigma)|)$.

Recall that if the step size is small enough, the intermediate results y_{ni} are defined by the explicit formulas

$$y_{ni} = y_n + h_n \sum_{j=1}^{i-1} a_{ij} f\bigl(x_{nj}, y_{nj}, S(x_{nj} - \tau)\bigr),$$

but if $h_n > \tau$, we may have some $x_{nj} - \tau_j > x_n$ so that $S(x)$ is evaluated in the span of the current step. In this situation we evaluate the formulas with simple iteration as described in the preceding section. Using this lemma we obtain a condition on the maximum step size that insures the convergence of simple iteration: Let $\delta^{(m)}$ be the maximum of $\|S^{(m+1)}(x) - S^{(m)}(x)\|$ for $x_n \leqslant x \leqslant x_{n+1}$. (The functions are the same for $x \leqslant x_n$.) Inequality (6) implies that $\delta^{(m)} \leqslant h_n \Gamma \delta^{(m-1)} \leqslant H \Gamma \delta^{(m-1)}$. From this it is obvious that simple iteration is contracting and the formula is well-defined if the maximum step size H is small enough to satisfy both (5) and

$$H\Gamma = HL \sum_{i=1}^{s} \max\bigl(|b_i(\sigma)|\bigr) \leqslant \frac{1}{2}. \tag{7}$$

We are now in a position to prove convergence as $H \to 0$ when the meshes include all discontinuities of low order and H satisfies (5) and (7). Suppose that the integration has reached x_n and let E_n be a bound on $\|y(x) - S(x)\|$ for all $x \leqslant x_n$. The local truncation error of the continuous extension is defined by

$$y(x_n + \sigma h_n) = y(x_n) + h_n \Phi\bigl(x_n, y(x_n), \sigma; y(x)\bigr) + lte_n(\sigma).$$

If we introduce

$$w_{n+\sigma} = y_n + h_n \Phi\bigl(x_n, y_n, \sigma; y(x)\bigr)$$

then inequality (4) and our assumption about the local truncation error imply that for $0 \leqslant \sigma \leqslant 1$,

$$\begin{aligned} \bigl\|y(x_n + \sigma h_n) - w_{n+\sigma}\bigr\| &\leqslant (1 + h_n \mathcal{L})\bigl\|y(x_n) - y_n\bigr\| + \bigl\|lte_n(\sigma)\bigr\| \\ &\leqslant (1 + h_n \mathcal{L}) E_n + C_1 h_n^{p+1}. \end{aligned}$$

The numerical solution is

$$y_{n+\sigma} = y_n + h_n \Phi\bigl(x_n, y_n, \sigma; S(x)\bigr).$$

The next step of the proof is complicated by the possibility that this formula is implicit. If we let Δ be the maximum of $\|y(x) - S(x)\|$ for $x \leqslant x_{n+1}$, then inequality (6) states that

$$\|y_{n+\sigma} - w_{n+\sigma}\| = h_n \bigl\|\Phi\bigl(x_n, y_n, \sigma; S(x)\bigr) - \Phi\bigl(x_n, y_n, \sigma; y(x)\bigr)\bigr\| \leqslant h_n \Gamma \Delta$$

and the triangle inequality implies that

$$\big\|y(x_n+\sigma h_n)-y_{n+\sigma}\big\| \leqslant (1+h_n\mathcal{L})E_n + C_1 h_n^{p+1} + h_n \Gamma \Delta.$$

This is a bound on $\|y(x)-S(x)\|$ for $x_n \leqslant x \leqslant x_{n+1}$ and by definition of E_n, the bound also holds for all $x \leqslant x_n$. We conclude that

$$\Delta \leqslant (1+h_n\mathcal{L})E_n + C_1 h_n^{p+1} + h_n \Gamma \Delta.$$

With the assumption that H satisfies (7), it then follows that

$$\Delta \leqslant \frac{1+h_n\mathcal{L}}{1-h_n\Gamma}E_n + \frac{C_1}{1-h_n\Gamma}h_n^{p+1} \leqslant \big(1+2(\mathcal{L}+\Gamma)h_n\big)E_n + 2C_1 h_n^{p+1}.$$

If we let $C_2 = 2(\mathcal{L}+\Gamma)$ and $C_3 = 2C_1$, this inequality tells us that

$$E_{n+1} = (1+h_n C_2)E_n + C_3 h_n^{p+1}$$

is a bound on $\|y(x)-S(x)\|$ for all $x \leqslant x_{n+1}$. Because we start with the given initial history, $E_0 = 0$. A simple induction argument shows that for all n,

$$E_n \leqslant C_3 \mathrm{e}^{C_2(x_n-a)}(x_n-a)H^p.$$

The uniform bound

$$E_n \leqslant C_3 \mathrm{e}^{C_2(b-a)}(b-a)H^p = CH^p$$

implies (3) for $a \leqslant x \leqslant b$.

4. Error estimation and control

Nowadays codes based on explicit Runge–Kutta triples estimate and control the error made in the current step by the lower order formula. They advance the integration with the higher order result y_{n+1} (local extrapolation) because it is more accurate, though just how much is not known. Our notation and proof of convergence incorporate this assumption about the triple. We make use of local extrapolation in our proof that we can estimate the local truncation error of the lower order formula, lte_n^*, by the computable quantity $est = y_{n+1} - y_{n+1}^*$.

The local truncation error

$$lte_n^* = \big(y(x_{n+1}) - y(x_n)\big) - h_n \Phi^*\big(x_n, y(x_n); y(x)\big)$$

is $\mathrm{O}(h_n^p)$. Using the corresponding definition of lte_n it follows easily that

$$lte_n^* = h_n\Phi\big(x_n, y(x_n); y(x)\big) - h_n\Phi^*\big(x_n, y(x_n); y(x)\big) + \mathrm{O}\big(h_n^{p+1}\big).$$

From the definition of y_{n+1} we have

$$h_n\Phi\big(x_n, y_n; S(x)\big) = y_{n+1} - y_n.$$

Using the convergence result (3), we show now that

$$h_n\Phi\big(x_n, y(x_n); y(x)\big) = h_n\Phi\big(x_n, y_n; S(x)\big) + \mathrm{O}\big(h_n H^p\big).$$

First we use inequality (4) to see that

$$\big\|h_n\Phi\big(x_n, y(x_n); y(x)\big) - h_n\Phi\big(x_n, y_n; y(x)\big)\big\| \leqslant h_n\mathcal{L}\big\|y(x_n)-y_n\big\| \leqslant h_n\mathcal{L}CH^p$$

and then inequality (6) to see that

$$\left\| h_n \Phi\big(x_n, y_n; y(x)\big) - h_n \Phi\big(x_n, y_n; S(x)\big) \right\| \leqslant h_n \Gamma C H^p.$$

The desired relation follows from the triangle inequality. The same argument establishes the corresponding relation for the lower order formula. With these two relations we have

$$\begin{aligned} lte_n^* &= h_n \Phi\big(x_n, y_n; S(x)\big) - h_n \Phi^*\big(x_n, y_n; S(x)\big) + \mathrm{O}\big(h_n H^p\big) \\ &= (y_{n+1} - y_n) - \big(y_{n+1}^* - y_n\big) + \mathrm{O}\big(h_n H^p\big) \\ &= est + \mathrm{O}\big(h_n H^p\big) \end{aligned}$$

which justifies our estimate of the local truncation error.

As with ODEs, the local truncation error can be expanded to

$$lte_n^* = \phi^*\big(x_n, y(x_n); y(x)\big) h_n^p + \mathrm{O}\big(h_n^{p+1}\big).$$

If we reject the step of size h_n from (x_n, y_n), we predict the local truncation error of another attempt of size h to be

$$est(h/h_n)^p \approx \phi^*\big(x_n, y(x_n); y(x)\big) h^p.$$

The same prediction applies to a step of size h from (x_{n+1}, y_{n+1}) on making the approximation $\phi^*(x_{n+1}, y(x_{n+1}); y(x)) \approx \phi^*(x_n, y(x_n); y(x))$ because the change in each of the arguments is $\mathrm{O}(h_n)$.

We control the local truncation error of y_{n+1}^*, but advance the integration with y_{n+1} because it is believed to be more accurate. Recall that for the BS(2,3) triple, the local truncation error of $y_{n+\sigma}$ attains its maximum for y_{n+1}. Accordingly, though we do not know the local truncation error of the continuous extension, we believe that it is smaller than required throughout the span of the step.

5. Tracking discontinuities

The constant lags $\tau_1, \ldots, \tau_k$ are supplied to `dde23` as the array `lags`. It is required that $\tau = \min(\tau_1, \ldots, \tau_k) > 0$ and that the delays are distinct. In simplest use the only discontinuity is at the initial point where the derivative obtained from the DDE, $y'(a+) = f(a, y(a), y(a - \tau_1), y(a - \tau_2), \ldots, y(a - \tau_k))$, is generally different from that provided by the history, $S'(a-)$. Generally the solution itself is continuous at the initial point, so we defer consideration of the case when it is not. It is characteristic of DDEs that this discontinuity in the first derivative results in discontinuities in the second derivative at $a + \tau_1, \ldots, a + \tau_k$. We describe this as discontinuities at the first level. In turn each of these discontinuities propagates in the next level as a discontinuity in the third derivative, and so forth. This is much simpler than the situation with more general DDEs because we can determine easily, and in advance, where the solution might have discontinuities and the lowest order discontinuity possible there. Some solvers ask users to supply details about which $y(x - \tau_j)$ appear in which equations so as to determine more precisely how discontinuities propagate. We think that the inconvenience of this design outweighs any advantages gained in the solution of the DDEs. Moreover, assuming that the worst can happen is more robust because there is no opportunity for a user to make a mistake in specifying the structure of the DDEs.

We have to track discontinuities and account for them during the integration because the usual order of a Runge–Kutta formula is seen in a step of size h_n from x_n only when $f(x, y(x), y(x - \tau_1), y(x - \tau_2), \ldots, y(x - \tau_k))$ is sufficiently smooth. As we have described the solution of DDEs with explicit

Runge–Kutta methods, each step is taken with two formulas in order to estimate the error made in the step. If the error estimation and step size adjustment algorithms are to work properly, f must be smooth enough for both to have their expected order. If we require that the step size be chosen so that each point of discontinuity is a mesh point x_m, then none of $y(x), y(x-\tau_1), \ldots, y(x-\tau_k)$ will have a low order discontinuity in the span of the current step. This is obvious for a discontinuity in the argument $y(x)$. There cannot be a ξ in (x_n, x_n+h_n) for which some $y(\xi-\tau_j)$ is not smooth because the discontinuity in y at $\xi-\tau_j$ would have propagated to ξ and we would have limited h_n so that $x_n+h_n \leqslant \xi$. Because we consider only one-step methods, it is only the smoothness of f in the current step that affects the order of the formula. This is true even if y or f is discontinuous at x_n provided that the correct values are used at x_n+, but we need take no special action in the solver because with one exception that we take up later, we assume these functions are (at least) continuous.

Although constant delays simplify greatly the location of potential discontinuities, there are some subtleties. One is revealed by a simple example. Suppose we start the integration at $x=0$ and the delays are $1/3$ and 1. The first lag implies a potentially severe lack of smoothness at $1/3, 2/3, 3/3, \ldots$. The difficulty is that in finite precision arithmetic, representation of $1/3$ is not exact, implying that the propagated value that ought to be 1 is merely very close to the correct value. Generally this is not important, but the lag of 1 implies a discontinuity precisely at 1. We must recognize that finite precision arithmetic has split two values that are the same else the requirement that the solver step to points of discontinuity will cause it to attempt steps that are too short to be meaningful in the precision available. In `dde23` we purge one of any pair of values that differ by no more than ten units of roundoff. This is done at each level of propagation so as to remove duplicates as soon as possible.

Example 4.4 of Oberle and Pesch [13] is a model for the spread of an infection due to Hoppensteadt and Waltman. The problem is posed on $[0, 10]$ and there is one lag, $\tau = 1$. The solution is continuous throughout the interval, but there are discontinuous changes in the differential equation at points known in advance. Indeed, the changes are qualitative because at first the equation does not depend on $y(x-1)$, hence is an ODE. We provide in `dde23` for discontinuities at points known in advance. All the user need do is give the locations of potential discontinuities as the value of the option `'Jumps'`. As with the ODE Suite, options for `dde23` are set with an auxiliary function, `ddeset`. In the case of the Hoppensteadt-Waltman model, the three points where discontinuities occur are defined by

```
    c = 1/sqrt(2);
    options = ddeset('Jumps',[(1-c), 1, (2-c)]);
```

and conveyed to the solver by means of the optional argument `options`. Discontinuities at known points are not limited to the interval of integration. The Marchuk immunology model discussed in [7, pp. 297–298] is solved on $[0, 60]$ and there is one lag, $\tau = 0.5$. One component of the history function is $V(t) = \max(0, t+10^{-6})$, which has a discontinuity in its first derivative at $t = -10^{-6}$. With the user interface of `dde23` it is easy to specify discontinuities at known points and they are accommodated in the algorithm simply by forming an array that includes these points as well as the initial point and then propagating them all as described already for the initial point. When using a solver with the capabilities of `dde23` it is easy and certainly better practice to specify discontinuities like those of these two examples, but a robust integrator may well be able to cope with them automatically.

Much more serious complications are introduced by continuation (restarts). As we discuss more fully in Section 6, we provide for the location of events. This capability is used, for example, to deal with a coefficient $\xi(m)$ in the Marchuk model that has a jump in its first derivative with respect to the state variable $m(t)$. Restarting the integration when this happens assures us that the formulas are applied only to smooth functions. Retaining the solution in the form of a structure makes it possible to continue. For the present we note only that we must include in the solution structure the list of potential discontinuities. After all, we might have discarded points outside the original interval of integration that lie in the span of the current interval. Accordingly, on a restart this list, any jumps specified by the user in the current integration, and the initial point of the current integration are formed into an array. The entries are propagated and duplicates purged as described earlier.

An initial discontinuity in the first derivative appears in derivatives of successively higher order. When the discontinuities are in a derivative of sufficiently high order that they do not affect the order of the formulas, we can stop tracking them. The relatively low order of the BS(2,3) triple implies that the effects of discontinuities are rather more short-lived than with other popular formulas. Our basic assumption is that $y(x)$ is continuous, so only four levels of propagation are needed. During the integration itself, the only time we permit a discontinuity in $y(x)$ is at the initial point, which is indicated by providing the value of the solution as the value of the option `'InitialY'`. The DDE is evaluated using this value, so both y and f are continuous to the right of the initial point. The facilities of `dde23` allow us to deal with discontinuous changes in the solution at other times. If the discontinuity occurs during the integration, as for example at the time of occurrence of an event, the integration is to be restarted with the appropriate value of the solution supplied via `'InitialY'`. Potential discontinuities of any order in the initial history are specified via the `'Jumps'` option. If either kind of discontinuity in $y(x)$ is possible, discontinuities are propagated to one higher level in the solver.

The Hoppensteadt–Waltman model involves the solution of ODEs as well as DDEs. The solver makes no assumption about whether terms with lags actually appear in the equations. This makes it possible to solve ODEs, though it is best then to set `lags = []` because any delay specified affects the list of potential low-order discontinuities, hence the details of the integration.

6. Event location

Recall that the Marchuk model has a coefficient $\xi(m)$ that has a jump in its first derivative with respect to the state variable $m(t)$. Such discontinuities are qualitatively different from the discontinuities that we treat with the `'Jumps'` option because they occur at unknown times. An event is said to occur at time t^* when a scalar function $g(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$, called an event function, vanishes at $t = t^*$. There may be many event functions and it may be important how the function vanishes, but we defer discussion of these complications. As with the propagated discontinuities, if we locate events and restart there, we integrate only with smooth functions. The theoretical and practical difficulties of event location are often not appreciated, but a theoretical analysis is possible with reasonable assumptions, e.g. [19], and a number of quality ODE solvers provide for the location of events. In particular, all the solvers of the MATLAB ODE Suite have a powerful event location capability that we have exploited in developing `dde23`. The capability is realized by means of the zero-finding function `odezero`. A nice exposition of its algorithm is found in [11]. Although we did not change the basic algorithm, we modified the function to account for the circumstances of `dde23`.

The user interface for event location in `dde23` is similar to that of `ode23`. There is no limit on the number of scalar event functions. They are all evaluated in a single function and the values returned as a vector. Using `ddeset`, the name of this function is passed to the solver as the value of the `'Events'` option. The function must also return some information about the nature of the events. Sometimes a user wants to know when events occur and the solution then, e.g., where a solution component has its local maxima and its values there. This is quite different from the events of the Marchuk model which cause the integration to be terminated. The two situations are distinguished using a vector `isterminal`. If we want to terminate the integration when event function k vanishes, we set component k of `isterminal` to 1, and otherwise to 0. There is an annoying matter of some importance: Sometimes we want to start an integration with an event function that vanishes at the initial point. Imagine, for example, that we fire a model rocket into the air and we want to know when it hits the ground. It is natural to use the height of the rocket as a terminal event function, but it vanishes at the initial time as well as the final time. `dde23` treats an event at the initial point in a special way. The solver locates such an event and reports it, but does not treat it as terminal, no matter how `isterminal` is set. It may be important how an event function vanishes. For example, to find local maxima of a solution component, we can locate zeros of the first derivative of this component. However, to distinguish maxima from minima, we want the solver to report a zero only when this function decreases through 0. This is done using the vector `direction`. If we are interested only in events for which event function k is increasing through 0, we set component k of `direction` to $+1$. Correspondingly, we set it to -1 if we are interested only in those events for which the event function is decreasing, and 0 if we are interested in all events.

It is hard to devise a good interface for event location because users may want to do quite different things when an event occurs. We have distinguished between terminal and non-terminal events. `dde23` returns its solution in the form of a structure. It can be called anything, but to be specific, suppose it is called `sol`. Often a user will want to know when an event occurred, what the solution is there, and which event function vanished. This information is returned by means of fields that have fixed names. If there are events, they occur at the entries of the vector `sol.xe`. For each entry, the solution there is the corresponding column of the array `sol.ye`. Also, the corresponding entry of the vector `sol.ie` is the index of the event function that vanished.

Often an event is terminal because it is necessary to change the function defining the equations, as with the Marchuk model, and/or the value of the solution before continuing the integration. The user interface for event location in `ode23` is powerful and reasonably convenient when solving ODEs because on a return from a terminal event, it is not difficult to make such changes, continue on as the solution of a new problem, and finally assemble the solutions on subintervals to obtain a solution on the whole interval of interest. The matter is much more difficult for DDEs because they involve previously given or computed quantities. On a restart, the solver may have to evaluate the given history function, which can be specified either as a function or a vector, as well as evaluate the solution computed prior to the restart. As mentioned in Section 5, propagated discontinuities also present complications. We deal with restarts by allowing the history to be specified as a solution structure. This structure contains all the information the solver needs to recognize and deal with the various possibilities. The approach is easy for the user, indeed, easier than when solving ODEs because the solution structure always contains the solution from the initial point to the last point reached in the integration.

As stated earlier, discontinuities in the solution are allowed only at the initial point of an integration. The solver is told of this and given the value of the solution there by providing it as the value of the option `'InitialY'`. Discontinuities in the solution are not common, and when they do occur, they invariably do so when a suitably defined event function vanishes. By making the event terminal, we can use the `'InitialY'` option to give the solution the correct initial value for continuation.

7. Long and short delays

The “natural” step size is the largest one that yields the specified accuracy. Codes for the solution of DDEs have difficulties when the delays are both long and short compared to the natural step size. Long delays present no special difficulty for `dde23`, so after we explain why, we give our attention to short delays.

On reaching a point x_n, the solver needs to evaluate the solution at previous points $x_n - \tau_j$, hence must retain the information it needs to do this. When there is a delay long compared to the natural step size, this implies that the solution must be retained over many steps. Many solvers are written in FORTRAN 77. Because this environment requires allocation of storage at the beginning of a run, there is a possibility of exceeding the storage provided. This is not an issue for `dde23` because we have followed the traditional design of ODE solvers in MATLAB of returning to the user the solution at all mesh points. Virtual storage and dynamic memory management increase the class of problems that can be solved, but we assume that long delays will not prevent solution of the problem given.

An ODE problem is stiff when the natural step size must be reduced greatly and often. Although the reduction necessary for the stability of explicit formulas gets the most attention, [16] points out reductions arising in the evaluation of implicit formulas by simple iteration and in output that might be just as severe. `dde23` uses simple iteration to evaluate the implicit formulas that arise when τ is smaller than the step size. The sufficient condition (7) for the convergence of simple iteration does not depend on τ, so delays cannot restrict the step size greatly for this reason. Some DDE solvers require that $H < \tau$ so as to have the simplicity of evaluating explicit formulas. For such solvers a short delay does cause stiffness. Another issue specific to DDEs is that solvers like `dde23` do not step over discontinuities. A short delay may cause the natural step size to be reduced greatly for this reason. However, this is not likely to happen so often when using `dde23` as to constitute stiffness. For one thing, the relatively low order of the formulas limits the number of steps affected by propagated discontinuities. For another, not stepping over discontinuities is different from limiting the step size to the shortest delay. To appreciate the distinction, suppose that there are two delays, $\tau_1 \ll \tau_2$. The first few steps of the integration are limited to τ_1 because there are potential discontinuities of low order at $a + \tau_1, a + 2\tau_1, \ldots$. The other delay makes its appearance first at $a + \tau_2$. Once the discontinuity at a that is propagating at spacing τ_1 has smoothed out, the solver can use step sizes much bigger than τ_1 in the rest of the integration to $a + \tau_2$. This all repeats at $a + \tau_2$ because the discontinuity there propagates at a spacing of τ_1 for a few steps and the second delay has its next effect at $a + 2\tau_2$.

There is an extensive literature on the stability of Runge–Kutta methods for DDEs, cf. [6,8,14,15] and references cited therein, much of it concerned with solving the scalar model problem

$$y'(x) = Ly(x) + My(x - \tau) \tag{8}$$

with constant step size h. Our investigation is somewhat novel in that we consider short delays and we relate the stability of our scheme when solving the model equation for DDEs to its stability when solving the test equation for ODEs, $y'(x) = Ly(x)$. Specifically, we prove that if the scheme is stable in a strong sense for the test equation, then for all sufficiently small τ and M, it is stable for (8). We conclude that in interesting circumstances, small delays do not affect adversely the stability of the scheme.

Suppose the integration has reached x_n and we take a step of size h. For all sufficiently small τ, the only argument of the BS(2,3) triple that does not lie in the span of the current step is $x_n - \tau$. It lies in $[x_n - h, x_n]$ where $S(x)$ is the cubic Hermite interpolant to $y_{n-1}, f_{n-1}, y_n, f_n$. As the form of the interpolant suggests and Maple [9] confirms,

$$S(x_n - \tau) = y_n - \tau f_n + \mathrm{O}(\tau^2). \tag{9}$$

The only way that y_{n-1} and f_{n-1} affect the computation of y_{n+1} is through the value $S(x_n - \tau)$. Because these quantities from the preceding step do not appear in the lowest order terms of the expansion of $S(x_n - \tau)$, we can investigate stability in the limit $\tau \to 0$ by considering only the current step. With this in mind, we can normalize by taking $y_n = 1$ so that for small τ, the integration is stable if $|y_{n+1}| < 1$.

In what follows we work only with terms through $\mathrm{O}(\tau^2)$. The quantity f_n is defined implicitly because $f_n = Ly_n + MS(x_n - \tau)$ and $S(x_n - \tau)$ depends on f_n. A little calculation using (9) shows that

$$f_n = \left(\frac{L + M}{M\tau + 1}\right) y_n + \mathrm{O}(\tau^2).$$

A similar expression holds for f_{n+1}. Using Maple it is now straightforward to determine how y_{n+1} depends on h, L, M, τ. Examination of the expression shows that it is easier to interpret if we introduce $z = hL$ and $Z = hM$.

Insight and a check on the computations is provided by supposing that $M = 0$. In this situation y_{n+1} should be, and is, a third order approximation to the solution of the test equation with $y_n = 1$, namely $y_{n+1} = P(z) = 1 + z + z^2/2 + z^3/6$. The polynomial $P(z)$ appearing here is the stability polynomial of the method for ODEs. The method is stable if $|P(z)| \leqslant 1$ and we shall say that it is “damped” if $|P(z)| < 1$.

So far we have been considering an expression for y_{n+1} that is useful for small τ. Let us now suppose that Z is also small. A straightforward computation using Maple shows that

$$y_{n+1} = P(z) + w_{12}(z)\, Z + \frac{w_{20}(z)}{h}\,\tau + \mathrm{O}(Z^2) + \mathrm{O}(\tau Z) + \mathrm{O}(\tau^2).$$

This expression makes clear that if for given h and L, the method is damped for the test equation, then it is stable for the model DDE (8) for all sufficiently small τ and M.

8. Examples

In this section we consider a pair of examples that show how easily even rather complicated DDEs can be solved with `dde23`.

8.1. Example 1

A Kermack–McKendrick model of an infectious disease with periodic outbreak is discussed in [7] where Fig. 15.6 shows the solution of

$$\begin{aligned} y_1'(t) &= -y_1(t)y_2(t-1) + y_2(t-10), \\ y_2'(t) &= y_1(t)y_2(t-1) - y_2(t), \\ y_3'(t) &= y_2(t) - y_2(t-10) \end{aligned} \tag{10}$$

on $[0, 40]$ with history $y_1(t) = 5,\ y_2(t) = 0.1,\ y_3(t) = 1$ for $t \leqslant 0$.

Solving this model illustrates the simplest use of `dde23`. In general a call to the solver with all options set by default has the form

```
sol = dde23(ddefile,lags,history,tspan);
```

The call list could scarcely be shorter. The interval of integration, `tspan`, is here `[0, 40]`. `history` can be specified in several ways. In the common case that the solution is constant prior to the initial point, the vector itself can be supplied as `history`. For the example it is `[5; 0.1; 1]`. The initial value of the solution is not an input argument for `dde23` because most problems have initial values that are the same as those given by the history function evaluated at the initial point. The constant delays are supplied as an array `lags`, here `[1, 10]`. There is no limit on the number of delays, but they must be distinct. `ddefile` is the name of the function for evaluating the DDEs. For the example this function can be coded as

```
function v = kmf(t,y,Z)
ylag1 = Z(:,1);
ylag2 = Z(:,2);
v = zeros(3,1);
v(1) = - y(1)*ylag1(2) + ylag2(2);
v(2) = y(1)*ylag1(2) - y(2);
v(3) = y(2) - ylag2(2);
```

Here the vector `y` approximates $y(t)$ and column j of the array `Z` approximates $y(t-\tau_j)$ for $j = 1, \ldots, k$. The delays can be specified in any order, but in defining the DDEs, τ_j is `lags(j)`. It is not necessary to create a local vector `ylag1` approximating $y(t-\tau_1)$ as we have done here, but we find that it often makes the equations easier to read.

Having coded the DDEs, we now solve the initial value problem with the command

```
sol = dde23('kmf',[1, 10],[5; 0.1; 1],[0, 40]);
```

The input arguments of `dde23` are much like those of `ode23`, but the output differs formally in that it is one structure, here called `sol`, rather than several arrays

```
[t,y,...] = ode23(...
```

The field `sol.x` corresponds to the array `t` of values of the independent variable returned by `ode23` and the field `sol.y`, to the array `y` of solution values. Fig. 15.6 of [7] is reproduced by

```
plot(sol.x,sol.y);
```

In summary, a complete program for solving the Kermack–McKendrick model consists of the file `kmf.m` defining the DDEs, the one command invoking `dde23`, and the one plot command.

Output from `dde23` is not just formally different from that of `ode23`. The method of `dde23` approximates $y(t)$ by a piecewise-polynomial function $S(t) \in C^1[a,b]$. The solver places in `sol` the information necessary to evaluate this function with `ddeval`. Values of $S(t)$ and optionally $S'(t)$ are obtained at an array of arguments t by

```
[S,Sp] = ddeval(sol,t);
```

With this form of output, we solve a DDE just once and then obtain inexpensively as many solution values as we like, anywhere we like. Plotting the solution at the mesh points of `sol.x` is generally satisfactory, but when the graph is not sufficiently smooth, we can always get a smooth graph by evaluating it at enough points with `ddeval`.

To further illustrate the advantages of output in this form, we explain how to reproduce a figure in [5]. A DDE with a single lag of 14 is solved there on $[0, 300]$. Fig. 2a plots $y(t-14)$ against $y(t)$, but starts with $t = 50$ so as to allow transient behavior to die out. The problem is solved easily with `dde23`, but we cannot make this plot using results at the mesh points `sol.x` alone. This is because they are not equally spaced: If t^* appears in `sol.x` so that we have $y(t^*)$ available in `sol.y`, it is generally not the case that $t^* - 14$ appears in `sol.x`, so we do not have $y(t^* - 14)$. However, using `ddeval` we can reproduce the figure easily with

```
t = linspace(50,300,1000);
y = ddeval(sol,t);
ylag = ddeval(sol,t - 14);
plot(y,ylag);
```

We specify 1000 points because generally a smooth graph in the phase plane requires a good many. Because `ddeval` is vectorized and exploits fast built-in functions, there is no need to be concerned about evaluating the solution at so many points.

As noted in Section 5, `dde23` does not require that the equations actually involve the delays specified. By specifying an additional, artificial delay that is short, the way the solution is computed is affected, but the solution and the natural step size remain the same. Solving the Kermack–McKendrick model with the option `'Stats'` set `'on'`, it is found that the problem is solved in 133 successful steps, with 17 failed attempts and a total of 451 function evaluations. Adding an artificial delay of 10^{-4} by changing `lags` to `[1, 10, 1e-4]` increased the number of successful steps to 164, the number of failed attempts dropped to 12, and the total number of function evaluations rose to 1027. If the step size had been restricted to the shortest delay, the integration would have required at least 40×10^4 steps!

8.2. Example 2

A two-wheeled suitcase may begin to rock from side to side as it is pulled. When this happens, the person pulling it attempts to return it to the vertical by applying a restoring moment to the handle. There is a delay in this response that can affect significantly the stability of the motion. This is modeled by Suherman et al. [20] with the DDE

$$\theta''(t) + \operatorname{sign}\big(\theta(t)\big)\gamma\cos\theta(t) - \sin\theta(t) + \beta\theta(t-\tau) = A\sin(\Omega t + \eta). \tag{11}$$

The equation is solved on $[0, 12]$ as a pair of first order equations with $y_1 = \theta$, $y_2 = \theta'$. Fig. 3 of [20] shows a plot of $y_1(t)$ against t and a plot of $y_2(t)$ against $y_1(t)$ when $\gamma = 2.48$, $\beta = 1$, $\tau = 0.1$, $A = 0.75$, $\Omega = 1.37$, $\eta = \arcsin(\gamma/A)$ and the initial history is the constant vector zero.

This complicated problem can be solved with a short program that we state now and then discuss:

```
state = +1;
opts = ddeset('RelTol',1e-5,'AbsTol',1e-5,'Events','scasee');
sol = dde23('scasef',0.1,[0; 0],[0, 12],opts,state);
```

```
while sol.x(end) < 12
  if sol.ie(end) == 1            % A wheel hit the ground.
    state = - state;
    opts = ddeset(opts,'InitialY',[0; 0.913*sol.y(2,end)]);
    sol = dde23('scasef',0.1,sol,[sol.x(end), 12],opts,state);
  else                           % The suitcase fell over.
    break;
  end
end
```

A parameter `state` is used for the value of $\mathrm{sign}(y_1(t))$. Parameters can be communicated as global variables or by adding them at the ends of the call lists. Using the latter technique, the `ddefile` can be coded as

```
function v = scasef(t,y,Z,state)
ylag = Z(:,1);
v = [ y(2); 0];
eta = asin(2.48/0.75);
v(2) = sin(y(1)) - state*0.248*cos(y(1)) - ylag(1) ...
      + 0.75*sin(1.37*t + eta);
```

A wheel hits the ground (the suitcase is vertical) when $y_1(t) = 0$ and the suitcase has fallen over when $|y_1(t)| = \pi/2$. All these events are terminal and all are to be reported. The event function can be coded as

```
function [value,isterminal,direction] = scasee(t,y,Z,state)
value = [y(1); abs(y(1))-pi/2];
isterminal = [1; 1];
direction = [0; 0];
```

When a wheel hits the ground, the integration is to be restarted with $y_1(t) = 0$ and $y_2(t)$ multiplied by the coefficient of restitution 0.913. The `'InitialY'` option is used for this purpose. The solution at all the mesh points is available as the field `sol.y` and in particular, the solution at the time of a terminal event is the last column of this array, `sol.y(:,end)`. The sign of `state` is changed to reflect the change in sign of $y_1(t)$. If the suitcase falls over, the run is terminated, so we must check which event occurred. This information is provided in `sol.ie` and it is the last event that we must check. Because we do not know how many events will occur, restarts are coded in a `while` loop.

Options are set by means of the auxiliary function `ddeset` exactly as with `odeset` when using `ode23`. In the first call to `ddeset`, we specify the event function and error tolerances. The call inside the `while` loop shows how to add an option and change the value of an option previously set.

The program reproduces Fig. 3 of [20] except that we have not included the standard commands that plot the figures. The event locations computed by `dde23` are compared in Table 1 to reference values computed with the FORTRAN 77 code DKLAG5 [12] used in [20] and verified with its successor DKLAG6 [3]. Having written the three solvers, we can fairly say that it is very much easier to solve this problem in MATLAB with `dde23`.

Table 1
Events for the suitcase problem

	dde23	reference
A wheel hit the ground at	4.5168	4.516757
A wheel hit the ground at	9.7511	9.751053
The suitcase fell over at	11.6704	11.670393

After running this program, `sol.xe` is

```
0.0000    4.5168    4.5168    9.7511    9.7511   11.6704
```

This does not seem to agree with the event locations of Table 1. For instance, why is there an event at 0? That is because one of the event functions is y_1 and this component of the solution has initial value 0. As explained in Section 6, `dde23` locates this event, but does not terminate the integration because the terminal event occurs at the initial point. The first integration terminates at the first point after the initial point where $y_1(t^*) = 0$, namely $t^* = 4.5168$. The second appearance of 4.5168 in `sol.xe` is the same event at the initial point of the second integration. The same thing happens at 9.7511 and finally the event at 11.6704 tells us that the suitcase fell over and we are finished.

References

[1] C.T.H. Baker, C.A.H. Paul, D.R. Willé, Issues in the numerical solution of evolutionary delay differential equations, Adv. Comput. Math. 3 (1995) 171–196.

[2] P. Bogacki, L.F. Shampine, A 3(2) pair of Runge–Kutta formulas, Appl. Math. Lett. 2 (1989) 321–325.

[3] S.P. Corwin, D. Sarafyan, S. Thompson, DKLAG6: A code based on continuously imbedded sixth order Runge–Kutta methods for the solution of state dependent functional differential equations, Appl. Numer. Math. 24 (1997) 319–333.

[4] J.R. Dormand, Numerical Methods for Differential Equations, CRC Press, Boca Raton, FL, 1996.

[5] J.D. Farmer, Chaotic attractors of an infinite-dimensional dynamical system, Phys. D 4 (1982) 366–393.

[6] N. Guglielmi, E. Hairer, Order stars and stability for delay differential equations, Numer. Math. 83 (1999) 371–383.

[7] E. Hairer, S.P. Nørsett, G. Wanner, Solving Ordinary Differential Equations I, Springer-Verlag, Berlin, 1987.

[8] K.J. in't Hout, On the stability of adaptations of Runge–Kutta methods to systems of delay differential equations, Appl. Numer. Math. 22 (1996) 237–250.

[9] Maple V Release 5, Waterloo Maple Inc., 450 Phillip St., Waterloo, ON N2L 5J2, Canada, 1998.

[10] MATLAB 5, The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, 1998.

[11] C.B. Moler, Are we there yet?—Zero crossing and event handling for differential equations, MATLAB News & Notes (1997) 16–17, available at http://www.mathworks.com/company/newsletter/pdf/97slCleve.pdf.

[12] K.W. Neves, S. Thompson, Software for the numerical solution of systems of functional differential equations with state dependent delays, Appl. Numer. Math. 9 (1992) 385–401.

[13] H.J. Oberle, H.J. Pesch, Numerical treatment of delay differential equations by Hermite interpolation, Numer. Math. 37 (1981) 235–255.

[14] C.A.H. Paul, C.T.H. Baker, Stability boundaries revisited-Runge–Kutta methods for delay differential equations, Numer. Anal. Rept. No. 205 (revised), Math. Dept., University of Manchester, UK, 1991.

[15] C.A.H. Paul, C.T.H. Baker, Explicit Runge–Kutta methods for the numerical solution of singular delay differential equations, Numer. Anal. Rept. No. 212 (revised), Math. Dept., University of Manchester, UK, 1992.
[16] L.F. Shampine, Numerical Solution of Ordinary Differential Equations, Chapman & Hall, New York, 1994.
[17] L.F. Shampine, Interpolation for Runge–Kutta methods, SIAM J. Numer. Anal. 22 (1985) 1014–1027.
[18] L.F. Shampine, M.W. Reichelt, The MATLAB ODE suite, SIAM J. Sci. Comput. 18 (1997) 1–22.
[19] L.F. Shampine, S. Thompson, Event location for ordinary differential equations, Comput. Math. Appl. 39 (2000) 43–54.
[20] S. Suherman, R.H. Plaut, L.T. Watson, S. Thompson, Effect of human response time on rocking instability of a two-wheeled suitcase, J. Sound Vibration 207 (1997).
[21] M. Zennaro, Delay differential equations: Theory and numerics, in: M. Ainsworth et al. (Eds.), Advances in Numerical Analysis: Theory and Numerics of Ordinary and Partial Differential Equations, Vol. IV, Clarendon Press, Oxford, 1995, pp. 291–333.