

Pulse Detection in Initial Value ODEs

Amy Marie Hynick¹ Patrick Keast^{a,1} Paul H. Muir^{b,1}

^a*Department of Mathematics and Statistics, Dalhousie University, Halifax, Nova Scotia B3H 3J5, Canada. (keast@mathstat.dal.ca)*

^b*Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, B3H 3C3, Canada. (muir@stmarys.ca)*

Abstract

In many physical models ordinary differential equations (ODEs) arise with the general form, $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}) + \mathbf{g}(t, \mathbf{y}(t))$, in which abrupt but large changes of limited duration, known as pulses, occur in $\mathbf{g}(t, \mathbf{y})$. These pulses may begin at times which are not known beforehand and may have unknown durations. If the duration is sufficiently short, standard differential equation solvers may miss the pulse completely, stepping over it, especially if, prior to the pulse, the solution is well behaved. In this paper we discuss software which employs standard initial value ODE software and a process of defect sampling to attempt to detect, and handle efficiently, any pulses which arise. The performance of the new software will be investigated by applying it to several test problems exhibiting pulses. The results show that pulses can be detected and efficiently handled by the new software and that significant computational savings are obtained.

Key words: pulse detection, initial value ordinary differential equations, defect sampling, efficiency.

AMS(MOS) subject classification: 65L05, 65L06.

1 Introduction

We consider systems of initial value ordinary differential equations (IVODEs) of the form

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) + \mathbf{g}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad (1)$$

¹ The work of these authors was partially supported by grants from the Natural Sciences and Engineering Research Council of Canada.

where $\mathbf{y} : \mathbb{R} \rightarrow \mathbb{R}^n$, $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, $\mathbf{g} : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, and $\mathbf{y}_0 \in \mathbb{R}^n$. It is assumed that \mathbf{f} is continuous and that \mathbf{g} is zero, except during some relatively short time period(s), whose duration and position may be unknown, and where it acquires an instantaneous and relatively large value. Such a sudden change will be called a *pulse*. Depending on \mathbf{f} , the underlying system of IODEs may be stiff or non-stiff.

To see the difficulty presented by systems such as (1), it is necessary to consider how most standard IODE solvers behave. Such a solver begins at t_0 and computes solution approximations at a set of points $t_i, i = 1, 2, 3, \dots$, where, usually, $t_i < t_{i+1}$. These points are selected by the software on the basis of a step selection algorithm which attempts to take steps that are as large as possible while keeping some estimate of the local error in each step within some user provided tolerance. The solution approximations will of course involve calculations based on evaluations of the right hand side of the ODE system given in (1). If, prior to a pulse being encountered, the solution of the IODE is well-behaved, showing no rapid changes, the software will increase the stepsize to improve efficiency while still satisfying the error requirements. The danger then, however, is that after a successful step *the code may step completely over the pulse*, and continue on the assumption that all is well, *even though a major feature of the solution has been missed*. An example of this kind of behavior, when the well-known IODE code LSODE [8,9] is used to attempt to solve a problem with a pulse, is given in Figure 4; LSODE computes a solution which shows no indication of a pulse, whereas the correct solution exhibits an obvious reaction to the pulse, as in Figure 3.

Two of the most critical quantities associated with a pulse are where it begins and how long it lasts; we will refer to these as the *start* and *duration* of the pulse. When both of these are known, we shall see that it is easy to force an IODE solver to detect the pulse and integrate through it in an efficient fashion. We shall refer to this as case (a). However, we are more interested in three other cases: (b) the start of the pulse is known but its duration is unknown, (c) the duration of the pulse is known but its start is unknown, and (d) neither the duration nor the start is known.

When the start of the pulse is unknown, simply detecting that a pulse is present becomes the central issue. The general approach we will use to detect a pulse is based on sampling the defect. In this approach, one assumes that in addition to providing a discrete numerical solution at the output points, t_i , the IODE solver also provides a continuous solution approximation (i.e., interpolant) over each step. The defect is defined to be the amount by which this continuous solution approximation fails to satisfy the IODE. That is, if $\mathbf{u}(t)$ is the continuous solution approximation at some point t , then the defect

at t is given by

$$\mathbf{r}(t) = \mathbf{u}'(t) - [\mathbf{f}(t, \mathbf{u}(t)) + \mathbf{g}(t, \mathbf{u}(t))]. \quad (2)$$

Note that this assumes that the IVIDE solver also provides the derivative of $\mathbf{u}(t)$. A measure of the quality of $\mathbf{u}(t)$ is obtained by sampling the defect, i.e. computing $\mathbf{r}(t)$, at a set of points within the current step.

When the underlying method upon which the IVIDE solver is based is a multistep method (see, e.g., [6]), the solution approximation at t_i is based on a computation in which the only evaluation of the right hand side of (1) within the current step, $(t_{i-1}, t_i]$, occurs at t_i . Thus if the pulse begins and ends strictly within this interval the solver has no way of noticing it. In the software described in this paper we will augment the calculation done by the IVIDE solver with a defect sampling process, which involves evaluating the defect and thus the right hand side of (1) at several points within $(t_{i-1}, t_i]$. When the duration of the pulse is known, we can guarantee that the pulse will be detected. Even when the duration is unknown, we can substantially improve the likelihood that the pulse will be detected. Once a pulse has been detected, it is important to the efficiency of the IVIDE solver that the pulse be handled in an appropriate way; i.e., since the beginning and end of the pulse look like discontinuities to the IVIDE solver, see, e.g., [7], it is important that our algorithms control the integration step sequence so that the IVIDE code steps into and out of the pulse efficiently.

We note that in some applications it will be possible for the user to determine, perhaps based on computed solution values as they are obtained, circumstances which will trigger a pulse. If this is possible, most IVIDE solvers will allow the user to handle the pulse using an *event location* option. Such an option allows the user to instruct the code to return whenever a pre-specified condition arises. Upon return in such a case, the user can then sharply reduce the stepsize and force the solver to correctly detect the pulse. We wish to emphasize that in this paper, we are assuming the more general case in which the user does not have sufficient a priori knowledge to predict the conditions which will cause a pulse to occur.

An important goal of our work is to develop software which can detect and efficiently treat pulses while employing an *unmodified* IVIDE solver. The key idea is that the algorithms and software presented in this paper are to be essentially independent of the IVIDE solver used to integrate the ODE system, provided the solver has several commonly available facilities (to be identified later in this paper). In this paper we consider coupling our software with an IVIDE solver based on a family of multistep methods, but with only minor modifications, our software could be employed, for example, with a one-step solver based on Runge-Kutta methods, provided that the solver employs a con-

tinuous extension of the Runge-Kutta formula to provide continuous solution and first derivative approximations over each step; see, e.g., [13].

This paper is organized as follows. In Section 2, we briefly review some related work. Section 3 considers the four different pulse cases mentioned above and describes a detailed algorithm for the efficient treatment of pulses within the context of these four cases. Section 5 presents and discusses our numerical results and Section 6 provides our summary, conclusions, and a discussion of future work.

2 Related Work

The most closely related work is in the area of handling of discontinuities in IVODE software. However, a fundamental difference is that in our context the primary problem is the actual detection of the pulse whereas in the treatment of discontinuities, detecting the presence of a discontinuity is only a secondary issue; the primary issues involve being efficient in precisely locating and stepping across the discontinuity. A standard IVODE code will often locate and step across a discontinuity with a large number of failed steps in a very inefficient way; see, e.g. [7], Figure 1. In that paper, Gear and Osterby discuss modifications to the stepsize and order selection algorithms of a variable order multistep code in order to allow it to efficiently locate and step over discontinuities in $\mathbf{f}(t, \mathbf{y})$ or its derivatives. Based on an estimate of the order of the discontinuity (i.e. the lowest derivative of $\mathbf{f}(t, \mathbf{y})$ that exhibits a discontinuity) and its magnitude, a sufficiently small step size in the region prior to the discontinuity is determined in order to allow the code to step over the discontinuity with a local error that will be within the user defined tolerance. The order of the discontinuity also dictates the order of the method used by the code as it steps over the discontinuity. Related work is discussed by Enright et al. [5] where the authors discuss modifications to an IVODE code based on Runge-Kutta methods to allow it to efficiently locate and step over discontinuities in $\mathbf{f}(t, \mathbf{y})$ or its derivatives. Within a step where a discontinuity is suspected, the main idea is to use several defect samplings, based on evaluations of the local high-order interpolant from the *previous* step, to accurately locate the discontinuity, and then to step over the discontinuity using an evaluation of the interpolant from the previous step. (The interpolant from the previous accepted step must be employed because no interpolant is available for the current step which has failed due to the presence of the discontinuity.)

While the discontinuity handling problem is of course related to the pulse detection and handling problem (in fact, for pulses of long duration, the pulse problem is equivalent to a pair of discontinuities), for pulses of short duration (which is a fundamental assumption of this paper), the primary issue, as men-

tioned above, is the actual detection of the pulse. When the start of the pulse is unknown a great percentage of the computational effort is associated with determining if a pulse is present within the current step taken by the IVODE code. Once this is determined, only a relatively minor cost is associated with accurately determining the start of the pulse (and the end of the pulse if the duration is also unknown).

3 Pulse Detection and Treatment

The pulse detection software employs a reliable IVODE solver which is used to solve the differential equation in a standard way except for interaction with the pulse detection algorithm at various times. Almost any IVODE program could be used, provided that it has certain features which the detection program requires. These features are:

- (i) The IVODE software should have the ability to restart at any time t with no information given about the solution or previous computation except the initial value of y at t . We call such a restart a *cold start*.
- (ii) It should be possible to request a return from the IVODE solver after every accepted step, with the possibility of continuing, using all the available information currently known about the solution. This will be called a *continuation*.
- (iii) After an accepted step, an interpolant should be available to enable the user to efficiently compute approximations to the solution and its first derivative at arbitrary points within the current step.
- (iv) It should be possible to specify a value of t , say t_{crit} beyond which the solver will not integrate. Without this feature the code might take a last step beyond the stopping value, and then interpolate to obtain the solution at this stopping value.

As mentioned earlier, the interpolant will be used to compute a defect. Our defect sampling approach is based on the following observation: *whenever an interpolant to a solution, computed using evaluations of the right hand side of the ODE system which are outside the pulse, is used to compute a defect as in (2), a large defect will be obtained for all samplings that take place within the pulse.*

We will call a defect *large* if, for some j ,

$$\frac{|r_j(t)|}{\max(1, |f_j(t, \mathbf{u}(t)) + g_j(t, \mathbf{u}(t))|)} > \frac{1}{2}, \quad (3)$$

where r_j , f_j , and g_j are the j -th components of $\mathbf{r}(t)$, $\mathbf{f}(t, \mathbf{u}(t))$, and $\mathbf{g}(t, \mathbf{u}(t))$, respectively, and t is the sample point. (This is somewhat conservative but it is consistent with the fact that we are using LSODE, a code which does not use local extrapolation - see, e.g. [13]. If we were to use an IVIDE solver that did employ local extrapolation, since the relative/absolute local error is controlled by tol , we could expect the relative/absolute defect to be at most some reasonably small multiple of tol - typical results, [3], in such a case put the defect within a factor of 2 of tol - and we could therefore replace the right hand side of (3) with, for example, $10 \times tol$, and have a tighter indicator of a large defect.)

The software we have developed can handle a sequence of non-overlapping pulses occurring in different components of the right hand side of (1); however for the purposes of clarity the following discussion will assume the presence of a single pulse.

There are three parameters defining a pulse:

- (i) t_c : the time at which the pulse begins.
- (ii) δ : the width or duration of the pulse interval, assumed to be small relative to the total time interval. We require, however, $\delta > t_c \cdot eps$, where eps is machine epsilon, thus ensuring that at least one machine number lies in the interval during which the pulse is active. For practical applications, we expect that the pulse width will actually be somewhat larger than this.
- (iii) $g_j(t, \mathbf{y}(t))$: for $j \in \{1, \dots, n\}$, the change in the j th component of the right hand side value contributed by the pulse, assumed to become large at t_c . That is, for some j , $g_j(t, \mathbf{y}(t))$ is assumed to be 0 for $t < t_c$ and for $t > t_c + \delta$, and is assumed to be large in magnitude for $t_c \leq t \leq t_c + \delta$.

We identify in more detail the four different cases distinguished by how much is assumed to be known about the pulse ($\mathbf{g}(t, \mathbf{y}(t))$ may use the known information; the unknown information may be dependent on the values of $\mathbf{y}(t)$ or on the results of some other computation).

- (a) The pulse begins at a known time and has a known duration, i.e. t_c and δ are both known.
- (b) The start time of the pulse is known, but the duration is unknown, i.e. t_c is known but δ is not.
- (c) The duration of the pulse is known, but it is not known when the pulse begins, i.e. δ is known but t_c is not.
- (d) Neither the start time nor the duration of the pulse is known, i.e. t_c and δ are both unknown.

We now provide an overview of our algorithm for handling these four cases.

Step 1, cases (a) and (b): The IVIDE solver is asked to integrate from t_0 to the known t_c value where the pulse begins.

Step 1, cases (c) and (d): As the integration proceeds, after every successful step control is returned from the IVIDE software to the pulse detection program. Let the time at the end of the current accepted step be t_{cur} and let the current stepsize be h_{cur} . Then we wish to sample at a number of points in the interval between the previously accepted time, $t_{prev} = t_{cur} - h_{cur}$ and t_{cur} . If δ is known, (case (c)), the number of sample points clearly should depend on it; we choose

$$n_s = \text{number of sample points} = \frac{h_{cur}}{\delta}s,$$

where s is an integer whose value may be set by the user, with a default value of $s = 2$. (As an alternative, assuming a *uniform* distribution of sample points, we can guarantee a sample point within the pulse by choosing

$$n_s = \alpha \frac{h_{cur}}{\delta},$$

for some $\alpha > 1$, removing the dependence on s .) If δ is unknown, (case (d)), the user can specify a number of sample points, n_s , (or the default of 20 points may be used). The sample points are chosen to be uniformly distributed across the current step.

After the defect has been evaluated at the sample points, the detection algorithm looks for any sample point for which a large defect is obtained. If all calculated defects are small, the integration continues with a continuation at t_{cur} . If the defect is found to be large at one or more sample points, then we find the smallest value of t , say t_{max} , with a large defect and the largest t , say t_{min} , less than t_{max} , that has a small defect - see Figure 1. We can then begin a bisection process to accurately locate t_c using $[t_{min}, t_{max}]$ as our interval.

The bisection algorithm we employ uses a relative tolerance of ϵ_{mach} to obtain a very accurate estimate for the beginning and end of the pulse. From the work of [7] and [5], it is clear that it is possible to save a few function evaluations by truncating this search earlier. As long as the distance to the beginning of the pulse is less than h , where h times the absolute value of the height of the pulse is less than the local error tolerance (in the appropriate relative/absolute sense), the code can step into the pulse while incurring a local error that is within the tolerance. A similar situation arises at the other end of the pulse. However, given the large number of function evaluations PODE already makes, we do not bother to introduce this extra complexity in our algorithm since the savings would be quite negligible. We note that by

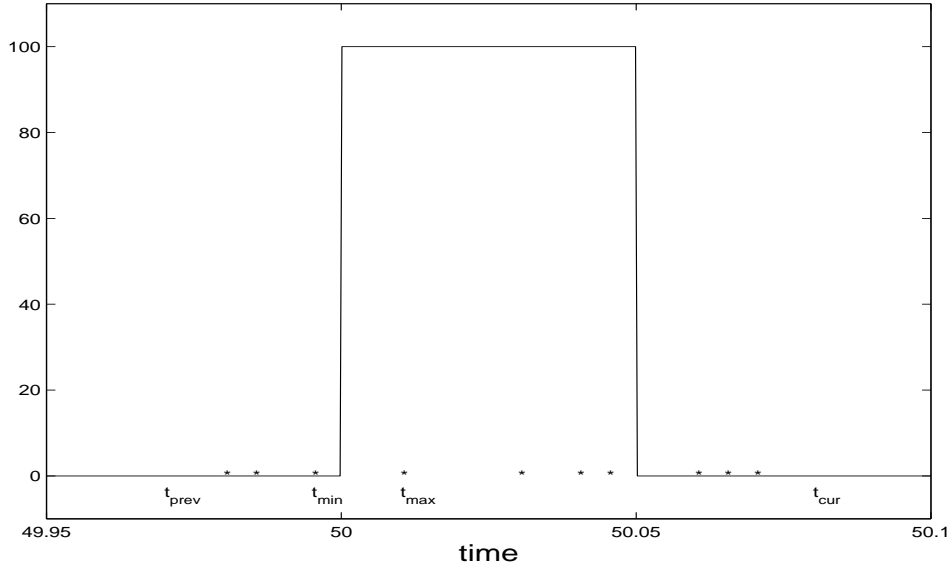


Fig. 1. Plot of time vs. defect for a step containing a pulse, with defect sample points labelled by *. Smallest sample point inside the pulse is t_{max} . Largest sample point to the left of the pulse is t_{min} .

choosing the bisection tolerance in this way we avoid having to ask the user to provide such a tolerance; this is helpful since most users would not know how to choose it appropriately.

We complete this part of the algorithm for cases (c) and (d) by evaluating the current interpolant at t_c to get a corresponding solution approximation.

Step 2, cases (a) and (c): We determine the end of the pulse from the known δ value.

Step 2, cases (b) and (d): We again use defect sampling within a bisection algorithm to accurately locate the end of the pulse. For case (b), we assume that the IODE solver has already integrated to t_c . We save the solution value at t_c and then ask the IODE code to take one more step with the stepsize prescribed by its step selection algorithm for the next step; since we assume that the pulse width is much smaller than a normal step, this step will carry it over the pulse. We can then employ defect sampling as described earlier. Assuming a sufficient number of sample points, this will allow us to determine some sample points with large defects, as in the previous subsection. Once this has been done, we can now treat cases (b) and (d) in the same way.

In order to locate the end of the pulse, we look for the largest sample point with a large defect, let us call it t_{max} , and the smallest sample point, greater than t_{max} , that has a small defect, say t_{min} - see Figure 2. Then $[t_{max}, t_{min}]$ is the appropriate interval for a bisection process that uses further defect sampling

to accurately locate the end of the pulse.

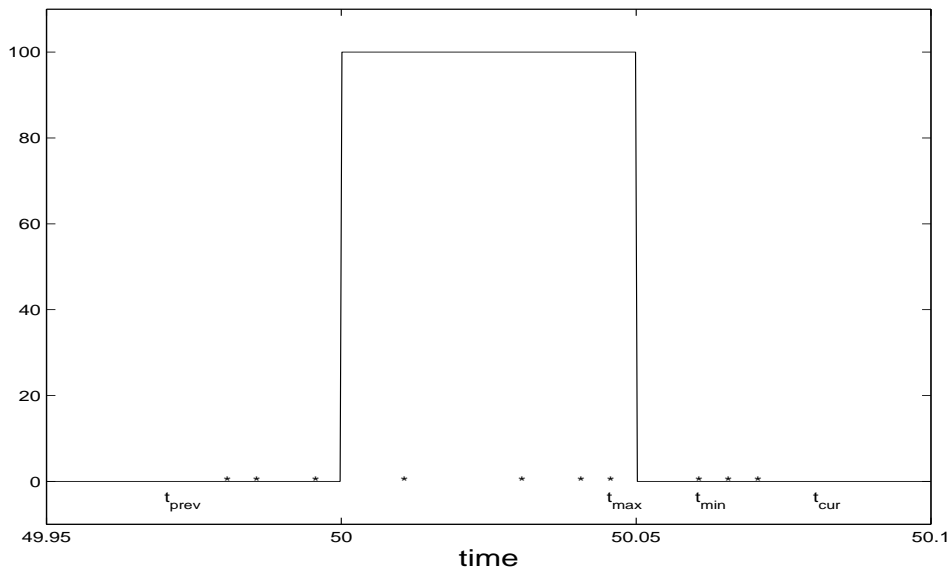


Fig. 2. Plot of time vs. defect for a step containing a pulse, with defect sample points labelled by *. Largest sample point inside the pulse is t_{max} . Smallest sample point to the right of the pulse is t_{min} .

Step 3: In all four cases we now have complete information: both t_c and δ are known. Also, in all four cases, the IVODE solver is now at t_c with a corresponding solution approximation to provide initial conditions for the next step. Thus the remainder of the algorithm is the same for all four cases. We restart the IVODE solver at the beginning of the pulse with a cold start, asking the code to integrate to the end of the pulse and then return control to the pulse handling software. After control is returned, the pulse detection software restarts the IVODE solver with a cold start, just beyond the end of the pulse.

In the implementation of this algorithm, when integrating to the beginning of the pulse, rather than taking $t_{crit} = t_c$, we actually take t_{crit} to be the largest machine number less than t_c . This prevents the IVODE solver from sensing a discontinuity at t_c and repeatedly incurring failed steps and halving the step size unnecessarily. Once inside the pulse, we integrate to $t_{crit} = t_c + \delta$. After exiting from the pulse, we restart at the machine number immediately after $t_c + \delta$.

4 Numerical Experiments

The above algorithms have been implemented in the code PDODE (*Pulse Detection in Ordinary Differential Equations*). The code, together with sample driving programs, may be obtained at <http://www.mscs.dal.ca/~keast/research/pulse>. The IODE solver employed within PDODE is LSODE [8,9]. This solver satisfies all the requirements of an IODE solver, required by our pulse detection code.

To demonstrate the efficacy of PDODE, we look at several test problems in which pulses occur. Although we do include this case in PDODE for convenience, we do not consider case (a) in this section; it is straightforward to get any standard IODE solver to handle this case efficiently: integrate to $t_{crit} = t_c$; perform a cold start at t_c with a return at $t_{crit} = t_c + \delta$; perform a cold start at $t_c + \delta$, integrating to t_{final} . We therefore use the test problems to allow us to compare the performance of a standard IODE solver (LSODE) with our pulse detection software, PDODE, for cases (b), (c), and (d). Recall that in case (b) we know t_c but not δ , in case (c) we know δ but not t_c , and in case (d) we know neither t_c nor δ .

We will see that for each problem both codes will obtain the correct solution. PDODE does this automatically but some intervention is required with LSODE so that it will not miss the pulse: in case (b), we run LSODE in its usual stepsize mode with $t_{crit} = t_c$ and then perform a cold start at t_c ; in case (c), we run LSODE with the maximum stepsize parameter set slightly less than the known pulse width; in case (d) LSODE must be run with a small maximum stepsize.

Two machine independent measures of the execution costs of ODE solvers are the required number of evaluations of the derivative, i.e., the right hand side function, $\mathbf{F}(t, \mathbf{y}(t)) \equiv \mathbf{f}(t, \mathbf{y}(t)) + \mathbf{g}(t, \mathbf{y}(t))$, and the required number of Jacobian evaluations, i.e. $\partial\mathbf{F}(t, \mathbf{y}(t))/\partial\mathbf{y}(t)$. The Jacobian evaluations involve $O(n^2)$ elements and more significantly lead to matrix computation costs that are $O(n^3)$. In the following experiments, we compare LSODE and PDODE with respect to these measures.

4.1 Problem 1

The first problem is *SB2* from [4], to which we have added a pulse term in the fourth ODE. The equations are

$$\begin{aligned} y_1' &= -10y_1 + 3y_2, & y_2' &= -3y_1 - 10y_2, & y_3' &= -4y_3, \\ y_4' &= -y_4 + P, & y_5' &= -0.5y_5, & y_6' &= -0.1y_6, \end{aligned} \tag{4}$$

where P is zero except in the range $50 \leq t \leq 50.005$ where $P = 100$. Thus $t_c = 50$ and $\delta = 0.005$. The initial conditions are $y_i(0) = 1, i = 1, \dots, 6$. This set of equations is stiff, which implies that LSODE must run in stiff mode [9]. We have $t_{final} = 100$ and relative and absolute tolerances of 10^{-10} . To obtain a good idea of what the actual solution component $y_4(t)$ looks like, we ran LSODE (very inefficiently) with a very small maximum stepsize of 0.0005. The resulting output is shown in Figure 3.

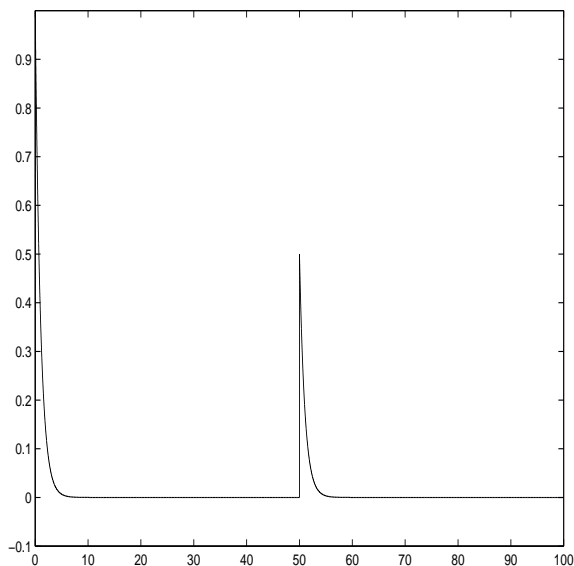


Fig. 3. Correct $y_4(t)$ for Problem 1

We begin our comparison by considering what happens when the above problem is given to LSODE, *with no restriction being placed on the stepsize*. Since the solution components decay rapidly to zero and LSODE employs a method suitable for stiff systems, the stepsize very quickly increases from about 10^{-11} at $t = 0$ to about 1 at $t = 50$, and consequently, *LSODE misses the pulse completely*. In Figure 4 the graph of the solution given by LSODE in this case is shown.

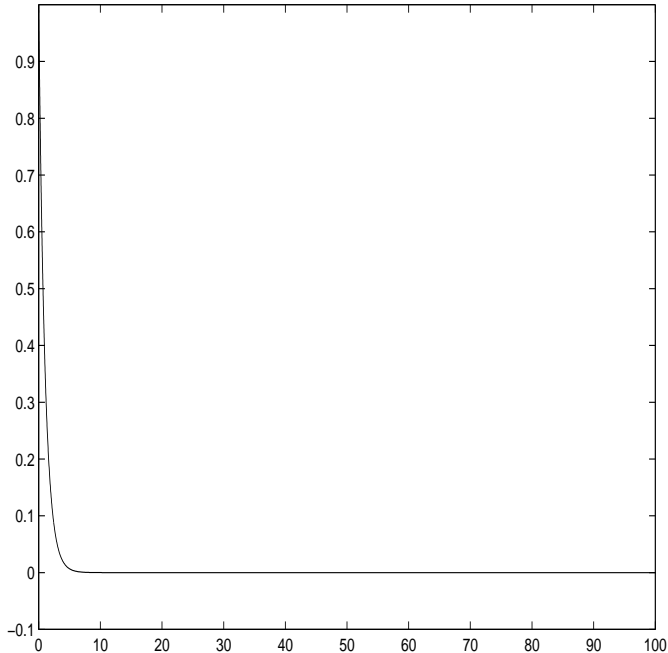


Fig. 4. $y_4(t)$, from LSODE in standard mode, for Problem 1, pulse missed

The results for LSODE and PODE under cases (b), (c), and (d) are given in Table 1. PODE finds the correct solution shown in Figure 3 and, with some intervention (as specified earlier), LSODE can also find it. From Table 1, we see that in case (b) the codes use about the same number of function evaluations but PODE uses about two-thirds as many Jacobian evaluations as LSODE. For case (c) PODE shows some improvement over LSODE in terms of evaluations of the right hand sides, with about 20% fewer function evaluations. More significantly PODE uses only about 5% of the number of Jacobian evaluations as LSODE does. For case (d) LSODE takes about four times as many function evaluations and more than a hundred times the number of Jacobian evaluations as PODE. (Since the pulse width is not known it is difficult to decide on an appropriate value of h_{max} for LSODE. If we were to experiment with $h_{max} = 10^{-r}$, $r = 1, 2, 3$, LSODE would not discover the pulse until $r = 3$. Then LSODE would use about 100,000 function evaluations and about 5000 Jacobian evaluations.)

4.2 Problem 2

4.3 Problem 3

A practical example of a system of IVODEs in which a pulse term appears arises in modelling the human heart, in work by Clements, Clements and

Method	Function calls	Jacobian calls
Pulse missed		
LSODE, default h_{max}	638	39
Case (b): Pulse detected		
LSODE, default h_{max}	983	99
PDODE, $s = 2$	921	68
Case (c): Pulse detected		
LSODE, $h_{max} = 0.004 < \delta$	25369	1370
PDODE, $s = 2$	20887	67
Case (d): Pulse detected		
LSODE, $h_{max} = 0.0005$	200296	10129
PDODE, $n_s = 100$	55055	67

Table 1
Function and Jacobian Evaluations for Problem 1

Horáček, [2], which employs the Luo–Rudy model [12] to investigate the phenomenon of re-excitation of the heart. One of many possible irregularities in the heart is a phenomenon that involves a wave of electrical activity reentering previously excited tissue, exciting it again. If this occurs repeatedly, it can lead to conditions in the heart conducive to cardiac arrhythmias. The Luo–Rudy model, which is based on the earlier work of [10], is associated with the study of electrical activity in the mammalian heart. To generate a heart beat, an electrical impulse is created by the sinoatrial node. This pulse excites cardiac muscle cells, causing them to contract. Even though the heart beats as a single unit, each cell of the heart does not contract at exactly the same time. Instead, the pulse travels throughout the heart in a wave-like fashion, via a stream of ions between individual cardiac cells, [11], [1]. In [12], a mathematical model of the action potential, or pulse, of a single mammalian ventricular cell is discussed.

The Luo Rudy model for the action potential of a single cardiac cell can be reduced to a system of eight coupled nonlinear ODEs. This system contains a number of constants and auxiliary functions, all of which are defined in [2];

the ODE system, with initial conditions, has the form,

$$\begin{aligned}
y_1' &= \left(I_{app}(t) - \sum_{k=1}^6 I_{ion,k}(\mathbf{y}) \right) / C_m, & y_1(t_0) &= -84.0 \equiv y_1^0 \\
y_2' &= (m_{inf}(y_1) - y_2) / \tau_m(y_1), & y_2(t_0) &= m_{inf}(y_1^0), \\
y_3' &= (h_{inf}(y_1) - y_3) / \tau_h(y_1), & y_3(t_0) &= h_{inf}(y_1^0), \\
y_4' &= (j_{inf}(y_1) - y_4) / \tau_j(y_1), & y_4(t_0) &= j_{inf}(y_1^0), \\
y_5' &= (d_{inf}(y_1) - y_5) / \tau_d(y_1), & y_5(t_0) &= d_{inf}(y_1^0), \\
y_6' &= (f_{inf}(y_1) - y_6) / \tau_f(y_1), & y_6(t_0) &= f_{inf}(y_1^0), \\
y_7' &= (X_{inf}(y_1) - y_7) / \tau_X(y_1), & y_7(t_0) &= X_{inf}(y_1^0), \\
y_8' &= -0.0001 I_{ion,2}(\mathbf{y}) + 0.07(0.0001 - y_8), & y_8(t_0) &= 0.0002,
\end{aligned} \tag{5}$$

where C_m , the membrane capacitance per unit area, is a constant, and the time-dependent pulse function, the applied current $I_{app}(t)$, defined as follows, ([2] consider several pulse cases; we provide an example here)

$$\begin{aligned}
I_{app} &= 0.0 \mu A/cm^2, & 0.0 \text{ ms} \leq t < 100.0 \text{ ms}, \\
I_{app} &= 55.0 \mu A/cm^2, & 100.0 \text{ ms} \leq t \leq 100.05 \text{ ms}, \\
I_{app} &= 0.0 \mu A/cm^2, & 100.05 \text{ ms} < t,
\end{aligned} \tag{6}$$

contributes a pulse to the first ODE; we have $t_c = 100.0$ and $\delta = 0.05$. The ionic current components, $I_{ion,k}(\mathbf{y})$, are nonlinear functions of \mathbf{y} and the remaining unidentified functions in (5) are nonlinear functions of y_1 ; see [2] for details.

By running LSODE with a very small maximum stepsize and a sharp tolerance, we were able to compute the correct form for y_1 , given in Figure 5. When this problem is presented to LSODE with no restriction on the stepsize, LSODE eventually begins to take large steps and completely misses the pulse. The approximation LSODE obtains for $y_1(t)$ is shown in Figure 6.

When PODE and LSODE are applied to this problem, PODE is able to detect the pulse automatically while LSODE can be made to detect the pulse with some intervention. Both codes return a solution as in Figure 5 with performance results as given in Table 2. These results show that for case (b) PODE uses about half the number of function evaluations and about one-quarter the number of Jacobian evaluations as LSODE, for case (c) LSODE uses about four times as many function evaluations and Jacobian evaluations as PODE, and for case (d) PODE uses about 3% of the number of function

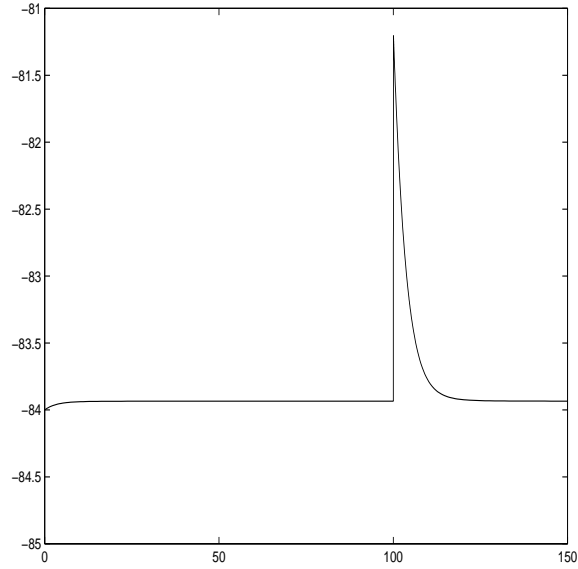


Fig. 5. Correct $y_1(t)$ for Problem 3

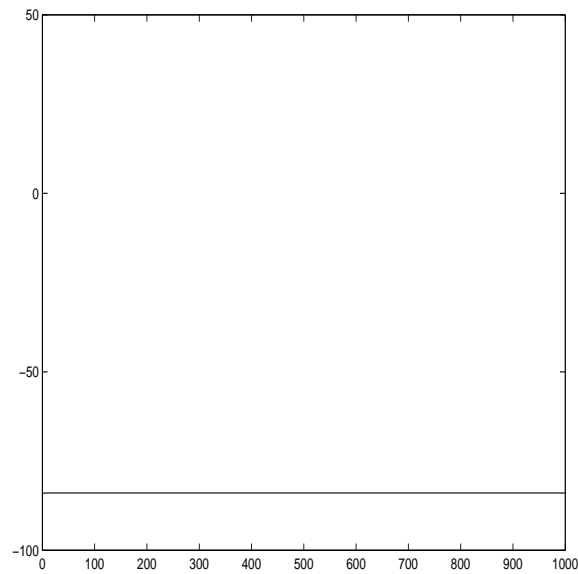


Fig. 6. $y_1(t)$, from LSODE in usual mode, for Problem 3, pulse missed evaluations and Jacobian evaluations of LSODE.

5 Summary, Conclusions, and Future Work

In this paper it has been shown through numerical experiments that standard IVIDE software will normally miss a pulse arising in the the right hand side function of an ODE system. To address this difficulty, this paper describes the design and implementation of a high-level algorithm called PODE that

Method	Function calls	Jacobian Evaluations
Pulse missed		
LSODE, default h_{max}	264	19
Case (b): Pulse detected		
LSODE, default h_{max}	1930	234
PDODE, $s = 2$	860	56
Case (c): Pulse detected		
LSODE, $h_{max} = 0.04 < \delta$	37066	1473
PDODE, $s = 2$	9675	372
Case (d): Pulse detected		
LSODE, $h_{max} = 0.005$	280431	10037
PDODE, $n_s = 20$	6447	316

Table 2

Function and Jacobian Evaluations for Problem 3

employs standard IVIDE software (LSODE in this paper) to efficiently detect and treat such pulses. A key advantage of the algorithm is that it is essentially independent of the underlying IVIDE code; thus, for example, LSODE could be replaced by an Runge-Kutta solver or by another multistep solver in a relatively straightforward fashion. PDODE has been shown to be reliable in finding and handling efficiently pulses for which either the start or the duration is not known, or when both are unknown. The most difficult case is when neither is known. In this case, LSODE has to have a severely restricted stepsize to ensure that it does not step over the pulse. PDODE typically uses fewer function evaluations than LSODE; more significantly, PDODE provides quite substantial savings on Jacobian evaluations, compared to LSODE. Since Jacobian evaluations are substantially more expensive than function evaluations this advantage for PDODE translates into large savings in overall execution time. (It should be noted that LSODE makes no effort to save Jacobian evaluations for future use; an IVIDE solver which was more conservative in its use of Jacobian evaluations would of course fair better than LSODE in a comparison with PDODE. However, it is clear that the severe stepsize restriction that must be placed upon any IVIDE solver in order for it to detect a pulse will imply that the solver will be more expensive than PDODE.)

The experiments presented in this paper have been run at a sharp relative/absolute tolerance with the hope that this would improve the ability of LSODE to detect the pulse without any interference in its step selection. As we have seen, even the use of a sharp tolerance does not help; experiments with coarse tolerances will of course reduce further the likelihood of LSODE finding a pulse.

The algorithm presented here for efficient pulse detection and handling is described assuming only one pulse in one component. However a sequence of non-overlapping pulses in multiple components represents no new challenge to the algorithm; it can simply be repeated over each integration region containing a pulse. In cases (a) and (b) this means that we simply integrate in standard mode to the next pulse; in case (c) and (d), we must integrate to the next pulse with our pulse detection algorithm in operation.

In terms of future work, it would be worthwhile to modify PDODE to incorporate a form of discontinuity detection and handling, using much of the approach we already have in place for pulses, as well as ideas from [7] and [5]. The possible presence of a discontinuity can be signalled by the unusual sequence of step failures and predictions as discussed in [7]. (Roughly speaking, this is a sequence of failed steps followed by a successful shorter step, with the next predicted step being large.) Since LSODE provides an interpolant for every step, we can use the interpolant from the last successful step to perform several defect samplings within the advance step where the discontinuity is suspected. This set of defect samplings can then be used to identify an interval containing the discontinuity, and bisection can then be used to locate it to sufficient accuracy.

Additional future work may involve improving the interface to PDODE to handle the multiple pulse case with less user intervention. Efficient handling of overlapping pulses in different ODE components also requires further investigation. As well, investigation of the performance of PDODE with IVIDE solvers other than LSODE might be interesting. The papers [7] and [5] also consider the case of discontinuities in higher derivatives of $\mathbf{f}(t, \mathbf{y})$; a related open question concerns whether the pulse detection algorithm can be generalized to detect and efficiently handle pulses arising only in some higher derivative of $\mathbf{f}(t, \mathbf{y})$.

References

- [1] American Heart Association, <http://www.americanheart.org>.
- [2] C.J. Clements, J.C. Clements, and B.M. Horáček, On the formation of scroll waves in an anisotropic ventricular myocardium, in: Shigui Ruan et al., eds., *Differential Equations with Applications to Biology* (Amer. Math. Soc., Providence, RI, 1999) 97–107.
- [3] W.H. Enright, Continuous numerical methods for ODEs with defect control, *J. Comp. Appl. Math.* **125** (2000) 159–170.
- [4] W.H. Enright, T.E. Hull, and B. Lindberg, Comparing numerical methods for stiff systems of ordinary differential equations, *BIT*, **15**, (1975), 10–48.

- [5] W.H. Enright, K.R. Jackson, S.P. Nørsett, and P.G. Thomsen, Effective solution of discontinuous IVPs using a Runge-Kutta formula pair with interpolants, *Appl. Math. Comp.* **27** (1988) 313–355.
- [6] C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1971).
- [7] C.W. Gear and O. Osterby, Solving ordinary differential equations with discontinuities, *ACM Trans. Math. Softw.* **10** (1984) 23–44.
- [8] A.C. Hindmarsh, ODEPACK, A Systematized Collection of ODE Solvers, in: R.S. Steplman et al., eds., *Scientific Computing* (North-Holland, Amsterdam, 1983) 55–64. (Available at <http://www.netlib.org>.)
- [9] A.C. Hindmarsh, Source for Module LSODE from Package ODEPACK, *NIST Guide to Available Math Software* (1987).
- [10] A.L. Hodgkin and A.F. Huxley, A quantitative description of membrane current and its application to conduction and excitation in the nerve, *J. Physiol* **117** (1952) 500–544.
- [11] J.S. Levine and K.R. Miller, *Biology: Discovering Life* (D.C. Heath and Company, Lexington, Massachusetts, 1991).
- [12] C.-H. Luo and Y. Rudy, A model of the ventricular cardiac action potential: depolarization, repolarization, and their interaction, *Circ. Res.* **68** (1991) 1501–1526.
- [13] L.F. Shampine, *Numerical Solution of Ordinary Differential Equations* (Chapman & Hall, New York, New York, 1994).
- [14] R.D. Skeel, Equivalent forms of multistep formulas, *Math. Comp.* **33** (1979) 1229–1250.