Appl. Math. Lett. Vol. 1, No. 2, pp. 153–156, 1988
Printed in Great Britain

0893-9659/88 $3.00 + 0.00
Pergamon Press plc

Locating Special Events when Solving ODEs

I. GLADWELL[1,2] , L. F. SHAMPINE[1,3]
R. W. BRANKIN

Mathematics Department, Southern Methodist University
Numerical Algorithms Group Ltd. Mayfield House

1 INTRODUCTION

Computing the solution, $y \epsilon R^n$, of the initial value problem in ordinary differential equations (ODEs),

$$y' = f(x, y), \qquad a \leq x \leq b, \; y(a) = y_a \tag{1.1}$$

may be only part of a larger task. Possibly the most common such task is to find either a first point $t_0 > a$ or a set of points $\{t_s\}$, $a < t_0 \leq t_1 \leq \ldots$, such that one of the equations

$$g_j(x, y(x), y'(x)) = 0, \qquad j = 1, 2, \ldots, m \tag{1.2}$$

is satisfied at $x = t_s$. The g_j are called "event functions", and event j is said to occur at t_s when t_s is a root of the jth event function. This note is concerned with problems that have the form either

$$g_j(x, y, y') \equiv y_{k_j}(x) - \alpha_j \tag{1.3}$$

or

$$g_j(x, y, y') \equiv y'_{k_j}(x), \tag{1.4}$$

where $y_{k_j}(x)$ is a component of the solution vector $y(x)$. Common problems such as

(i) finding where a component of the solution assumes a given value, and
(ii) finding where a component of the solution has an extremum

have the form (1.3) or (1.4) with a single event function. We allow several event functions of both forms at the same time, and so can solve more complicated problems such as

(iii) tabulating values of a dependent variable y_j;
(iv) determining the location of switching points or points of discontinuity defined in terms of linear functions of a dependent variable y_j; and
(v) determining zeros of a general event function $q(x, y) = 0$ by adjoining a differential equation for q to the system (1.1).

We outline an approach to finding all the event locations for these special event functions. In addition to emphasizing the difficulty of the problem in general, we indicate how we construct an event locating code and how we *graft* such a code onto a standard integrator with an interpolation feature. Full details of the analysis, the codes we have developed, and numerical examples are given in [1,2].

[1]Partially supported by NATO Research Grant 898/83
[2]Partially supported by a Royal Society/SERC Industrial Fellowship and by the Numerical Algorithms Group Ltd.
[3]Partially supported by the Applied Mathematical Sciences program of the Office of Energy Research under DOE grant DE–FG05–86ER25024

2 SOME DIFFICULTIES WITH THE PROBLEM

Popular codes for the initial value problem (1.1) step from a to b, producing approximations y_i, y_i' to $y(x_i), y'(x_i)$ at a set of points $a = x_0 < x_1 < \ldots$. It is usual with such codes to test each event function of (1.2) for different signs at x_i and x_{i+1}. A change of sign in any one indicates that an event has occurred in $[x_i, x_{i+1}]$. Adams and backward differentiation formula codes produce a polynomial $p(x)$ that approximates $y(x)$ on all of $[x_i, x_{i+1}]$. It is natural then to compute the first root of

$$g_j(x, p(x), p'(x)) = 0 \tag{2.1}$$

as an approximation to the location of event j. The popular Runge-Kutta formulas produce solutions only at the mesh points x_i. This approach to the event location problem has been an important reason for the recent work aimed at providing these formulas with polynomial approximate solutions.

The way of locating events just described is so natural that many have been led to think that the task itself is easy. This is far from true. The mesh points x_i are chosen to provide efficiently approximations to $y(x)$ and $y'(x)$ of a specified accuracy. Because the event functions (1.2) do not influence the selection of the mesh, the spacing may not be at all appropriate for locating the positions of the events, t_*, that is, the roots of (1.2). If an even number of roots, counting multiplicity, should occur between x_i and x_{i+1}, they will not be noticed. Should the presence of an event be noticed, there is in general no way to be certain the root-solver will find the *first* root.

It may not be possible to locate an event accurately when solving (1.2) for a root τ. There is error in the root due to the fact that the function is not evaluated perfectly. How sensitive the root is to errors in the function evaluation is a statement about its conditioning. It is not unusual in the present context to encounter ill-conditioning. What is special now is that we actually solve (2.1) for a root ω as an approximation to τ. It is not generally possible to compute τ accurately because we solve the wrong problem (namely (2.1) instead of (1.2)), rather than because the problem solved is ill-conditioned (though it may be!). If we do not integrate the differential equation accurately, we cannot locate the root accurately.

3 A ROOT-FINDING ALGORITHM

The key observation is that for event functions of the form (1.3) and (1.4), when $y(x)$ is approximated by a polynomial $p(x)$, then $g_j(x, p(x), p'(x))$ is itself a polynomial. Using Sturm sequences we can, in principle, answer the question, "Is there a root of the polynomial $g_j(x, p(x), p'(x))$ in the interval $(x_i, x_{i+1}]$?". We can be sure of computing the *first* such root by combining, for example, bisection and Sturm sequences to test for its presence.

We need to be clear what we mean by the "first position" at which equation (1.2) is satisfied. It is quite possible, and indeed fairly common in practice, that (1.2) is satisfied at the *initial* point $a = x_0$. Of course, the user is able to check this, and so there is no need to report it. On each call to the routine for integrating (1.1), the code steps from the current point x_i to an internally chosen point x_{i+1}. We search this interval for the next occurrence of an event defined by (1.2). Now if we define the current integration interval as the *half open* interval $(x_i, x_{i+1}]$, then overall we lose no points from the range of integration, except the initial point $x_0 = a$. This definition has two significant advantages. First, our technique for determining the roots is based on a Sturm sequence algorithm and the count of the number of zeros given by this algorithm is always defined on such an interval. Second, if we locate a root at a point $t \epsilon (x_i, x_{i+1}]$ and we wish to go on to locate the position of the next event in the direction of integration, then the interval to be searched initially is naturally $(t, x_{i+1}]$.

Because we want our algorithm for locating the first position of any event to be both efficient and robust, we have tried to devise an algorithm that converges rapidly when we are satisfied that we have reduced the task to locating a root for a single event function, and is cautious in other less frequently occurring circumstances. If we did not insist on this efficiency, we might employ a simpler algorithm. For example, we could apply a standard code for computing the roots of polynomials to locate all occurrences of all events and then rank the results. This approach would be palpably inefficient even if we were first to check which events occur in $(x_i, x_{i+1}]$. Important to efficiency is that in many applications there will be just one event function, that is, $m = 1$ in (1.2). Also, we expect that in many integration intervals, there will be no events at all; that when an event does occur, it will usually be the only one in the interval; and that if more than one event occurs in one integration interval, the events will be isolated.

4 Combining Root-Finder and Integrators

Rather than rewrite popular integrators to add a root-finding capability, we have taken the novel approach of producing a root-finder that can be grafted onto many integrators. We assume only that the integrator can be used in a mode such that it returns to the calling program after each step from x_i to x_{i+1} with a polynomial of known degree r representing the solution on all of $[x_i, x_{i+1}]$. As we explain in [2] anomalous behavior is possible with the interpolants of certain popular codes because they do not connect smoothly at mesh points. This does not interfere with the root-finder in its treatment of the interval $(x_i, x_{i+1}]$, and provided the user appreciates the potential for difficulties arising from the lack of smoothness, there is no reason not to use our root-finder with such a code.

The modules we have written for the root-finding task assume that the polynomial interpolant has a specific form. Popular integrators represent their interpolating polynomials in many forms, so a conversion routine is needed. In [2] we consider how to do this when no information about the form is supplied, just the ability to evaluate the interpolant. It is hardly surprising that a better job can be done when more information is supplied. In [1] we present a collection of subroutines implementing the algorithm described in this note. Normally no evaluations of the ODE are required in our algorithm over and above those required for the integration alone. After an interval containing an event is located, our algorithm is nearly as efficient as the root finder on which it is based when the latter is used in the standard way to calculate a root of a single equation.

In addition to the computed event locations our codes return "condition numbers" and an estimate of the multiplicity of the root. Examples of the use of the code for both routine and pathological problems are found in [1,2]. One such example is the problem of tabulating x at equispaced steps in the solution y of the (artificially constructed) equation

$$y' = -y^2 + x^6 - 2x^5 + x^4 + 3x^2 - 2x, \; y(-1) = -2.$$

The solution is $y = -x^2(1 - x)$. We tabulate the values of x at which $y = -1, 0, 1, 2$ using the NAG code D02NBF [**NAG**] as the integrator with relative local error tolerance 10^{-5}. One way to proceed is to define just one event function at a time, and after the position of each event has been located, to redefine the event function. Alternatively, we can use *four* event functions corresponding to all *four* tabulation points simultaneously. In Table 1 we present the values found with the latter approach and an error estimate based on the assumption that the integration error, $y_j(\tau) - p_j(\tau)$, is about equal to the local error tolerance. Though not well established, the error estimate is clearly a reasonable approximation to the true error at $x = 1$, but an overestimate for the error of the approximations to the double zero at $x = 0$. The results for the approach using just one event function are exactly the underlined values in Table 1.

y	Tabulation Point	Multiplicity	Error Estimate
-1	-0.07549	1	$0.808E-4$
0	-0.00022	1	0.135
0	0.00023	1	0.135
0	0.99999	1	0.555E-4
1	1.46557	1	$0.158E-4$
2	1.69562	1	$0.106E-4$

Table 1:Tabulation points for equispaced values of y.

REFERENCES

1. R.W. Brankin, I. Gladwell, and L.F. Shampine, *Codes for Reliable Solution of Special Event Location Problems for ODEs*, Num. Anal. Rept. **139** (1987), Dept. of Math., University of Manchester.
2. I. Gladwell, L.F. Shampine, and R.W. Brankin,, *Reliable Solution of Special Event Location Problems for ODEs*, Num. Anal. Rept. **138** (1987), Dept. of Math., University of Manchester.
3. "NAG Fortran Library Manual," Mark 12, Numerical Algorithms Group Ltd., Oxford, U.K., 1987.

Mathematics Department, Southern Methodist University, Dallas, Texas 75275
Numerical Algorithms Group Ltd. Mayfield House, 256 Banbury Road, Oxford OX2 7DE, United Kingdom