A Friendly Fortran DDE Solver

S. Thompson
Department of Mathematics & Statistics
Radford University
Radford, VA 24142
thompson@radford.edu

L.F. Shampine
Mathematics Department
Southern Methodist University
Dallas, TX 75275
lshampin@mail.smu.edu

March 22, 2004

This manuscript is a preprint of a paper prepared for Volterra 2004, The Third International Conference on the Numerical Solution of Volterra and Delay Equations.

Abstract

DKLAG6 is a FORTRAN 77 code widely used to solve delay differential equations (DDEs). Like all the popular Fortran DDE solvers, new users find it formidable and in many respects, it is not easy to use. We have applied our experience writing DDE solvers in MATLAB and the capabilities of Fortran 90 to the development of a friendly Fortran DDE solver. How we accomplished this may be of interest to others who would like to modernize legacy code. In the course of developing a completely new user interface, we have added significantly to the capabilities of DKLAG6.

1 Introduction

This investigation is another step in a line of software development that began with the first ordinary differential equation (ODE) solver, `ode45`, of the problem solving environment (PSE) MATLAB [7]. `ode45` is a translation of the FORTRAN solver RKF45 [15], but it is not merely a translation: C.B. Moler defined a new user interface to make solving ODEs as easy as possible. In this he exploited the capabilities of the MATLAB language and recognized the kinds of problems typical of the PSE. Later Shampine and Reichelt [12] extended this interface to increase the capabilities of the software and accommodate other

methods for solving ODEs. In particular, all the solvers of the ODE Suite were endowed with a powerful event location capability. Programs for solving delay differential equations (DDEs) have been *much* harder to use than programs for solving ODEs, in part because the task is itself more complicated. We firmly believe, however, that more scientists would use DDE models if it were easier to solve them numerically. This led us to develop the solver `dde23` [14] that is now part of MATLAB. It is restricted to problems with constant delays, but given that, it makes solving simple problems easy and it solves conveniently problems with complications like event location and restarts. The `ddesd` program [10] solves DDEs with time- and state-dependent delays. Though the underlying method is quite different from that of `dde23`, the user interface is as similar as the tasks allow. For brevity we refer to these MATLAB DDE solvers as `dde23/sd`.

The DDE solver DKLAG6 [3] has capabilities that are comparable to those of `dde23/sd` and solves a larger class of problems, but new users find it formidable and in many respects, it is not easy to use. We believe that it is fair to say this about all the popular DDE solvers written in FORTRAN. Our principal goal was to develop a program, DDE_SOLVER, based on DKLAG6 that approaches the convenience of `dde23/sd`. DKLAG6 is written in FORTRAN 77, but to reach our goal, we had to have the capabilities of Fortran 90/95. For brevity we write F77 to indicate FORTRAN 77 and F90 to indicate Fortran 90/95. A friendly user interface is already enough to justify our effort, but we have also made important qualitative improvements to DKLAG6. One example is the way storage is handled. The F77 code requires a user to supply some work arrays. It is not known in advance how big these arrays must be and if they are not big enough, the computation fails. By exploiting the dynamic storage possibilities of F90, we do not have to trouble users about storage at all. A cursory examination of the Netlib software repository at www.netlib.org shows that much of the quality mathematical software found there is written in F77. Here we show by example how F90 can be exploited to make legacy software both easier to use and more powerful.

We are mainly interested in solving retarded DDEs for which the (vector) solution $y(t)$ satisfies

$$y'(t) = f(t, y(t), y(\beta_1), \ldots, y(\beta_k)) \tag{1}$$

for $t_0 \le t \le t_f$. The terms $y(\beta_j) = y(\beta_j(t, y(t)))$ are the solution evaluated at delayed arguments. They are delayed because we suppose that the delay functions β_j all satisfy $\beta_j(t, y(t)) \le t$. The problem is said to be singular at t^* if a delay goes to 0 there, i.e., $\beta_i(t^*, y(t^*)) = t^*$ for some i. Such problems present both theoretical and computational difficulties. Although DDE_SOLVER can solve some singular problems, we are mainly interested in retarded DDEs for which the delayed argument is always less than t. A solution must also satisfy

$$y(t) = h(t) \tag{2}$$

for $t < t_0$ and a given history function $h(t)$. A relatively common kind of problem with singular delay functions is a so-called initial value DDE. Such

a problem requires only the initial value of the solution because all the delay functions satisfy $\beta_j(t, y(t)) \geq t_0$ for all t. Although DDE_SOLVER allows time- and state-dependent delay functions, the most common form involves a constant lag $\tau_j > 0$, namely

$$\beta_j(t, y(t)) = t - \tau_j \tag{3}$$

DDE_SOLVER makes special provision for problems with constant lags because they are very common, they are considerably easier to solve, and the user interface can be simplified.

Neutral DDEs depend on values of the derivative of the solution at delayed arguments. They are difficult in both theory and practice. DDE_SOLVER allows equations of the form

$$y'(t) = f(t, y(t), y(\beta_1), \ldots, y(\beta_k), y'(\beta_1), \ldots, y'(\beta_k)) \tag{4}$$

In addition to (2), the solution of a neutral equation must also satisfy

$$y'(t) = h'(t) \tag{5}$$

for $t < t_0$. It is easy enough to attempt the solution of a neutral problem with DDE_SOLVER and the legacy code has performed well in practice, but these problems can be very difficult and success is problematic.

Along with the integration of the DDEs, we may be interested in locating where any one of a collection of event functions,

$$g_j(t, y(t), y'(t)) \tag{6}$$

vanishes. The places where they vanish are called events. Sometimes we just want the solver to report the location of an event and the solution there. Other times we want to terminate the integration or restart the integration after changing the problem. Not very many DDE solvers provide for event location. DKLAG6 does, and we have augmented its capabilities for DDE_SOLVER.

In §10 we illustrate the solution of an equation of retarded type used to model lasers. The one delay function involves a constant lag and the history is constant. There is an event function. DDE_SOLVER is much faster than `dde23` and not much harder to use. Other aspects of DDE_SOLVER are illustrated by a problem of neutral type. The delay function is both time- and state-dependent and it is singular. The history is not constant. Despite these serious difficulties, DDE_SOLVER solves the problem easily.

DDE_SOLVER and a collection of example programs are available at the web site [5]. The typical program writes the solution to a file. In most cases the F90 program is accompanied by an M-file that imports the numerical solution and plots it in MATLAB.

2 Simple Calls for Simple Problems

One of our goals was to make solving simple problems as easy as possible. As it has turned out, all you have to do is define the problem: A problem of the

form (1,2,3) is solved with a call like

```
SOL = DDE_SOLVER(NVAR,DDES,BETA,HISTORY,TSPAN)          (7)
```

Here NVAR is an integer array of two entries. The first is NEQN, the number of equations in (1), and the second is NLAGS, the number of delays in (3). DDES is the name of a subroutine for evaluating (1). It has the form

```
SUBROUTINE DDES(T,Y,Z,DY)                               (8)
```

The input arguments are the independent variable T, a vector Y of NEQN components approximating $y(T)$, and an array Z that is NEQN $\times$ NLAGS. Column j of this array is an approximation to $y(\beta_j(T, y(T)))$. The subroutine is to evaluate (1) with these arguments and return $y'(T)$ as the vector DY of NEQN components. This minimal call list is simpler than that of DKLAG6:

```
SUBROUTINE DDES(NEQN,T,Y,Z,DYLAG,DY,RPAR,IPAR)          (9)
```

The number of equations is not needed and in F90 there are easier ways to pass parameters than by array arguments RPAR and IPAR that are not used for a typical problem. The DYLAG argument is even less commonly used because it is meaningful only when solving neutral problems.

It is not obvious from the call lists (8) and (9), but there is a major difference in the form of the DDEs solved by DKLAG6 and DDE_SOLVER. Following the lead of Neves [8], DKLAG6 requires that each solution component determine exactly one delay (i.e., NLAGS = NEQN). In contrast, DDE_SOLVER accepts equations of the form (1) that allows each equation to depend on all the delays. It is not hard to introduce new unknowns to put a general system of DDEs into the special form required by DKLAG6, but it is some trouble for the user. One reason the special form has not proved popular is that preparation of the problem generally involves introducing several new unknowns that are pseudonyms for one of the original unknowns. This means that the numerical solution returned by the solver contains several approximations to the same function, a considerable annoyance when manipulating the numerical solution. Though the special form has some technical advantages, we believe that they are far outweighed by the convenience of the general form (1).

BETA is the name of a subroutine for evaluating the delays and HISTORY is the name of a subroutine for evaluating (2). To be more precise, the functions are defined by subroutines for general problems, but they are defined in a simpler way in the very common situations of constant lags and/or constant history. How this is done is explained in §4. In DDE_SOLVER the user writes a subroutine to evaluate all components of the history (2). This contrasts with the approach of DKLAG6 which calls the history subroutine with a specific value for j and the user has to code the subroutine so as to evaluate just component j of the history. We believe that it is more convenient for users simply to evaluate all the components, especially when there are many.

Output is handled very differently in DKLAG6 and DDE_SOLVER. Both codes have a number of possibilities, but here we describe only the default

NVAR	Number of DDEs, delays, and event functions (vector)
DDES	Subroutine to evaluate DDEs
DELAYS	Subroutine for delays (vector for constant lags)
HISTORY	Subroutine for initial history (vector for constant history)
TSPAN	Interval of integration and output points

Table 1: Required input arguments to DDE_SOLVER.

modes. DKLAG6 returns the numerical solution at times equally spaced in the interval of integration. DDE_SOLVER returns the numerical solution in the output structure SOL. The input vector TSPAN is used to inform the solver of the interval of integration and where approximate solutions are desired. TSPAN has at least two entries. The first entry is the initial point of the integration, t_0, and the last is the final point, t_f. If TSPAN has only two entries, approximate solutions are returned at all the mesh points selected by the solver itself. These points generally produce a smooth graph when the numerical solution is plotted. If TSPAN has entries $t_0 < t_1 < \ldots < t_f$, the solver returns approximate solutions at (only) these points. The number and placement of these output points has little effect on the cost of the integration. That is because the solver selects a mesh that allows it to compute an accurate solution efficiently and evaluates a continuous extension (a polynomial interpolant) to obtain approximate solutions at specified output points.

Integration statistics are available as components of a vector field in SOL. They are the number of successful steps, the number of failed steps, the number of derivative evaluations, the number of failed corrector iterations, the total amount of array storage used, and the total number of root finding functions used. A convenient way to inspect these statistics is to display them using an auxiliary subroutine: CALL PRINT_STATS(SOL)

The call list of (7) resembles closely that of `dde23/sd`. It is scarcely more complicated when the DDEs are of neutral type. We defer a discussion of the matter to §7 because a neutral problem is treated as an option. Our design provides for a considerable variety of additional capabilities. This is accomplished in two ways. F90 provides for optional arguments that can be supplied in any order if associated with a keyword. This is used, for example, to pass to the solver the name of a subroutine EF for evaluating (6) with a call like

```
SOL = DDE_SOLVER(NVAR,DDES,BETA,HISTORY,TSPAN,EVENT_FCN=EF)
```

Details are provided below. One of the optional arguments is a structure containing options. This structure is formed by a function called DDE_SET that is analogous to the function `ddeset` used by `dde23/sd`. The call list of (7) uses defaults for important quantities such as error tolerances, but of course, the user has the option of specifying quantities appropriate to the problem at hand. These options are discussed in sections that follow.

3 New Answers to Old Questions

In this section we explain how a number of the software issues in DKLAG6 can be handled in a more satisfactory way by exploiting F90. Other convenient aspects of F90 are illustrated by the numerical examples of §10 where quantities are initialized where they are declared and arrays are formed using array constructors. These examples also show the use of IMPLICIT NONE to disable implicit typing in Fortran.

3.1 Precision

MATLAB does all computations in a working precision. This contrasts with Fortran for which it is necessary to select a precision for REAL variables. DKLAG6 is coded for double precision. At the time DKLAG6 was written, double precision arithmetic could have a quite different meaning on computers in wide use. A generally accepted way to deal with this was to call a subroutine D1MACH to obtain the hardware characteristics, and that is what is done in DKLAG6. The most common possibilities are listed in the subroutine and on installing DKLAG6, the subroutine has to be edited so as to return values appropriate for the target machine. F90 handles this in a much more satisfactory way by providing intrinsics for the purpose in the language itself. For instance, the unit roundoff corresponding to double precision is obtained by UROUND = EPSILON(1D0). F90 makes it possible to code software so that the precision can be changed easily, but this complicates the way that users have to write their subroutines. Also, the IEEE arithmetic standard has become generally accepted and with it, double precision is by far the usual choice for scientific computing. For these reasons DDE_SOLVER is coded entirely in double precision and it assumes that the user will provide subroutines coded in a familiar way using double precision.

3.2 Array Operations

DKLAG6 makes use of several of the BLAS [6]. These F77 subroutines were developed to clarify vector operations by encapsulating them and to speed up the operations by means of implementations that exploit the hardware. Their functionality is now directly available in F90. For instance, a vector K2 of length N is multiplied by a scalar ALPHA and added to a vector R of the same length by CALL DAXPY(N,ALPHA,K2,1,R,1). With the array operations of F90 this can be expressed in a more straightforward way as R = ALPHA*K2 + R. This feature allowed us to replace numerous loops throughout the code with much simpler statements.

3.3 Storage

A basic difficulty in writing mathematical software in F77 is to manage storage in a way convenient for users. We address the issue of not knowing in advance how

much storage will be needed in the next section and here take up only provision of a fixed amount of storage. DKLAG6 requires a good many vectors of working storage that have a length known in advance, generally NEQN or NLAGS. These vectors are of no interest to the user, so making them arguments in the call list of DKLAG6 would make the call list intolerably long. The standard way of dealing with this difficulty was to have the user supply a work array with a length given by a formula involving NEQN (and possibly other parameters). Segments of this array are used for the various vectors of the algorithm. If this were done directly, the algorithm would be unreadable, so pointers are defined on entry to DKLAG6 and segments of the work array are used as arguments in calls to lower level subroutines. The lower level subroutines have all the requisite vectors in their call lists. In this way the vectors of the algorithm all have meaningful names, but the user is exposed only to the work array at the top level. The call lists of the lower level subroutines are exceedingly long: D6DRV3 has 47 arguments, the core intergrator has 55 arguments, and several subroutines have 35 or more! This is a satisfactory way to proceed, but the user still has to provide one or more work arrays and it is easy to make mistakes when modifying/maintaining the program. This can all be handled more conveniently in F90 by means of global storage. In the specification part of the module for the solver we declare the various vectors as allocatable arrays. This makes them visible to all the subroutines of the module. By giving them the PRIVATE attribute, they are not visible outside the module. To be fair, something similar could have been done in F77 using labelled COMMON, but F90 furnishes a more disciplined approach to global data. Using global arrays we have simplified the program notably by doing away with the passing of a great many vectors through several levels of codes. None of the DDE_SOLVER subroutine call lists has as many as 20 arguments and most have considerably fewer. For example, D6DRV3's call list of 47 arguments was reduced to 13 and the core integrator's call list of 41 arguments was reduced to 8.

3.4 Dynamic Storage

An obvious difference between DKLAG6 and DDE_SOLVER is that the user does not have to consider storage at all. This is much more than a mere convenience; it is a qualitative improvement in the software. That is because it is not known in advance how much storage will be needed for some of the arrays. One example is the queue that holds the discontinuity tree—the number of discontinuities depends on the problem. A person using DKLAG6 must guess how much storage will be required and if the guess is not big enough, the computation will fail. Another example is output. Values of the solution at the mesh points selected by a quality solver generally provide a smooth graph, but none of the popular Fortran solvers returns these values because the number of mesh points is not known in advance.

F90 has dynamic storage capabilities and array operations that we use in DDE_SOLVER to deal with the work arrays and queue. In the first instance this is a matter of making these arrays ALLOCATABLE and an initial allocation

of enough storage to solve an easy problem. Of course it might turn out that more storage will be needed to complete the computation. The difficulty then is that we need to increase the sizes of arrays whilst retaining the values stored in the arrays. Though a bit clumsy, this be done in a straightforward way using temporary storage. For reasons of efficiency, we increase the sizes of the arrays by a substantial amount. If we could return an allocatable array, we would proceed similarly with the solution itself, but that is not allowed in F90. Instead we define a derived type DDE_SOL with fields like

```
DOUBLE PRECISION, DIMENSION(:,:), POINTER :: Y
```

and return a structure SOL of this type. In the solver we can allocate storage for the field SOL%Y and store the solution in it as the integration proceeds. Like the working storage we increase the storage as necessary in chunks of substantial size, but before returning we trim the array to the size actually needed for the solution.

4 Constant Functions

In general the initial history is defined by a function, but a constant history is common in practice. For this reason DDE_SOLVER allows users to supply a constant history in the form of a vector rather than making them write a subroutine that returns a constant vector. Similarly, a user must generally supply a function that evaluates the delayed arguments $\beta_j(t, y(t))$. However, it is common in practice that all the delayed arguments have the form $t - \tau_j$ for constant lags τ_j. For such problems it is convenient to supply the vector of lags and have the solver form the delayed arguments as needed. Constant lags greatly simplifies construction of the discontinuity tree and in other ways leads to a more efficient and accurate integration. The DKLAG6 of [3] did not exploit this possibility, but DDE_SOLVER does.

Simpler calls in common circumstances are achieved in F90 by writing several versions of the solver that users call with the same (generic) name, DDE_SOLVER. In F90 the number and type of input arguments in a given call are compared to the various versions and the matching version of the solver is selected for execution. All our versions of DDE_SOLVER call the same underlying code, a modification of DKLAG6. If, say, HISTORY is supplied as a vector, the version of the solver that expects a vector argument is selected. In this way we make the solution of DDEs rather easier in the common circumstances of constant history and/or constant lags.

5 Optional Arguments

F90 allows optional arguments to functions and subroutines that follow the required arguments in the call list. Though optional arguments can be passed by position in the list, we assume that they will always be identified with keywords,

OPTIONS	Options structure formed with DDE_SET
EVENT_FCN	Subroutine to evaluate event functions
CHANGE_FCN	Subroutine for action at an event
OUT_FCN	Subroutine for output

Table 2: Optional input arguments to DDE_SOLVER.

another new possibility in F90. Using keywords, we can set just the options of interest and we can set them in any order. This was illustrated in §2 with the EVENT_FCN option. The options that are handled in this way are discussed below. We begin with specification of a variety of options using an options structure and the optional argument OPTIONS.

5.1 Options

To simplify the solution of DDEs, many of the options allowed by `dde23/sd` are grouped in an options structure. The structure is an optional argument to the solver and if it is not present, defaults are used for all the options. The options structure is formed by an auxiliary function, `ddeset`. In an invocation of this function, the user sets any options he wishes and any options not set are given default values. It is possible to do something quite similar in F90. We have defined a derived type DDE_OPTS with fields for the various options. The auxiliary function DDE_SET is used to set the options. The PRESENT intrinsic is used inside DDE_SET to determine whether the user has set an optional argument and if not, a default value is given to the field. An options structure is an optional argument in DDE_SOLVER. All arguments of DDE_SET are optional, so if the user does not provide an options structure to the solver, it will use OPTIONS = DDE_SET() to form an options structure with all fields set to default values. As a straightforward example of the use of options we mention a guess HINIT for the initial step size. This is an input argument of DKLAG6, but it is an option for DDE_SOLVER. The new solver selects a value automatically by default and if the user specifies a value as in OPTS = DDE_SET(HINIT=1D-7), it will try the specified value. We have moved a good many arguments from the call list of DKLAG6 to the options structure. This is convenient, but what is far more important is that the user has to set only those options for which default values are inappropriate for the problem at hand.

We discuss more fully the error tolerances because they are the options most commonly set by the user and they present some points of interest. DKLAG6 has relative and absolute error tolerances that are vectors. A vector relative error tolerance was in vogue at the time DKLAG6 was written, but since then it has become generally accepted that a scalar relative error tolerance is simpler and perfectly adequate, so that is what we have in DDE_SOLVER. Tolerances must be specified for DKLAG6, but they are optional arguments in the call to DDE_SET. If the scalar relative error tolerance RE is not specified, it is given a default value of 10^{-3}. Absolute error tolerances are more complicated.

RE	Relative error tolerance
AE	Absolute error tolerance
AE_VECTOR	Vector of absolute error tolerances
ISTERMINAL	Specify terminal events
DIRECTION	Distinguish how event functions cross axis
INTERPOLATION	Prepare to interpolate solution
NEUTRAL	Solve DDE of neutral type
HINIT	Initial step size
HMAX	Maximum step size
JUMPS	Discontinuities at points known in advance
TRACK_DISCONTINUITIES	Disable tracking of discontinuities
TRACKING_LEVEL	Maximum discontinuity tracking level

Table 3: Options set with DDE_SET

The most common case is the default, which is to set all NEQN components of the absolute error tolerance vector to 10^{-6}. There is no analog of this case in DKLAG6, but the second most common case is to use the same value for all components. It is a temptation to forego the remaining case of a vector tolerance, but this is not possible because a vector tolerance is actually needed for a good many problems. To facilitate the second case, DKLAG6 allows users to specify a vector of only one component. However, to do this the user must also give an input argument ITOL a value that instructs DKLAG6 to expand this vector to one of length NEQN. The inconvenience of setting ITOL can be eliminated in F90 by using the SIZE intrinsic to recognize the case, but declaring a vector tolerance of length one is unnatural when solving a system of NEQN equations. This is handled better in `dde23/sd` because MATLAB does not distinguish a scalar and a vector of one component. We have not been able to do the same in F90, so we have resorted to *two* options for absolute error tolerances in DDE_SET. Corresponding to the scalar RE option there is a scalar AE option. In addition there is a vector option called AE_VECTOR. If no absolute error tolerance is specified, the default is used. If the scalar option AE is set, the value input is assigned to all NEQN components of an absolute error tolerance vector used by the solver. If the vector option AE_VECTOR is set, the vector (of NEQN components) that is input is used for the absolute error tolerance vector. If both options are set inadvertently, the more detailed vector option is given precedence. Proceeding in this way, the more common the case of absolute error tolerances, the easier it is to specify.

5.2 Event Location

In our experience event location has been crucial to the effective solution of many problems. It is not widely available, certainly not in DDE solvers, in part because the task is difficult. Some of these difficulties are discussed in [11, 13]. We considered event location to be an essential capability for DKLAG6 [3].

It was a explicit design goal for the MATLAB ODE Suite [12], so there was never any question that `dde23/sd` would have it. The MATLAB solvers have a more powerful capability for event location than DKLAG6, so we provided their additional features in DDE_SOLVER. In most respects the interface for event location in DDE_SOLVER resembles closely that of `dde23/sd`. In particular, both solvers return the same information about events, viz., the location, the numerical solution at the event, and an integer identifying which event occurred, as fields in the solution structure. However, the event functions (6) allowed by DDE_SOLVER are more general than those of `dde23/sd` in that they can depend on $y'(t)$. And, some matters are handled in a different way.

Event location is optional, so the solver must be told that it is to locate zeros of a collection of functions evaluated in a given procedure. In `dde23/sd` the user does this by providing a function handle as the value of the option `Events` using `ddeset`. In DDE_SOLVER the user provides the name of the subroutine as an optional input argument of the solver itself that is specified with the keyword EVENT_FCN.

It can be useful to distinguish how an event function crosses the axis. For example, this can be used to deal with problems that have events at the initial point. In the physical example of §10 it is used to distinguish a maximum from a minimum when the first derivative of a solution component is an event function. In `dde23/sd` the function for evaluating events must return a vector that states what the user wants. Component j of a vector `direction` is given the value -1 when event j is interesting only if the event function decreases through 0, the value $+1$ when it is interesting only if the function increases through 0, and the value 0 when it does not matter how the function crosses the axis. We added this capability to DDE_SOLVER, but the vector DIRECTION is set as an option using DDE_SET. It has a default value with all components equal to 0. This is different from `dde23/sd`, which does not have a default for `direction`. Adding this feature to the new solver was not merely a matter of adding it to the interface; DKLAG6 does not have this feature, so the algorithm for locating events had to be augmented appropriately.

A very important distinction is made between terminal and non-terminal events. As the name suggests, a terminal event causes the program to finish up and return control to the user. In `dde23/sd` the function for evaluating events must return a vector `isterminal`. A value of 1 in component j means that the integration is to terminate if the event function (6) crosses the axis in the specified direction, and a value of 0 means that information about the event is to be made available, but the integration is to continue. Coding in F90 it was more natural to make the vector ISTERMINAL of type LOGICAL. Also, this vector is not optional in `dde23/sd`, but it has a default value with all components equal to .FALSE. in DDE_SOLVER.

DKLAG6 and `dde23/sd` take fundamentally different approaches to terminal events. To understand the issues, a concrete example will be helpful. In [14] we discuss the solution of DDEs modelling the behavior of a two-wheeled suitcase [16] with DKLAG6 and `dde23`. As the suitcase rolls along, it can start to rock from one wheel to the other. When a wheel hits the ground, the integration

must be restarted with new initial conditions that reflect the wheel bouncing, i.e., with a change of direction and a speed reduced by a coefficient of restitution. In MATLAB it was convenient to store as fields in the solution structure returned by `dde23/sd` all the information needed to restart from the last point reached. Coding this is complicated because of all the possibilities, but from a user's point of view, restarting is merely a matter of supplying a solution structure as history rather than a function (or vector). We can solve the suitcase problem in this design by making a wheel hitting the ground a terminal event. If the solver returns because a wheel hit the ground, we restart the integration with initial conditions determined appropriately from the solution returned at the time of the event.

DKLAG6 does not provide for restarts and after careful consideration, we decided not to add this capability to DDE_SOLVER. One reason is that it would be a major change to the legacy code. Another is that it is not clear to us how to code something using F90 that resembles closely what we did in `dde23/sd`. Finally, the approach taken in DKLAG6 is quite satisfactory; it is just that the user interface is unduly complicated. In our new design the user writes a subroutine of the form

```
CHANGE(NE,T,Y,DY,HINIT,DIRECTION,ISTERMINAL,QUIT)
```

and informs the solver of this subroutine by means of an optional input argument with keyword CHANGE_FCN. The solver calls this subroutine at every event. In this call NE is an integer that identifies which event occurred, T is the location of the event, and Y and DY are the approximations to the solution and its first derivative, respectively, at the event. In CHANGE the user can inspect this information and decide what, if any, action is appropriate. For instance, he might want to redefine the DDEs. It is good practice to define the DDEs and auxiliary subroutines such as CHANGE and EVENTS in a module. This makes possible variables that are available to all the subroutines of the module. It is then easy to use such a variable in the DDEs subroutine to determine how the equations are to be evaluated and to change the value of this variable in CHANGE in response to an event. Y and DY are also output variables that the user can reset. That is just what is needed to solve the suitcase problem. In effect we restart the integration with new initial values. The output variable HINIT allows a user to specify a step size for the solver to try on return. The integer array DIRECTION and the logical array ISTERMINAL can be changed so that the event functions will be interpreted differently on return from CHANGE. The integration can be terminated by changing QUIT from its input value of .FALSE. to .TRUE.

Using event functions it is possible to increase the efficiency of the solver for some problems. One such problem is C2 from the test suite in [4]. The DDES are

$$\begin{aligned} y_1'(t) &= -2y_1(t - y_2(t)) \\ y_2'(t) &= |y_1(t - y_2(t))| - \frac{|y_1(t)|}{1 + |y_1(t - y_2(t))|} \end{aligned}$$

The problem is to be solved on the interval $[0, 40]$. On this interval, there are more than 50 discontinuities caused by sign changes in the delayed solution and the solution. Ignoring these discontinuities and solving the problem with error tolerances of 10^{-5}, we found that there were 513 successful steps and 466 unsuccessful steps. Using the two event functions

$$\begin{aligned} g_1 &= y_1(t - y_2(t)) \\ g_2 &= y_1(t) \end{aligned}$$

and derivative switching flags ([13]) reduces the cost to 334 successful steps and 138 unsuccessful steps. The accuracy of the two solutions was comparable, but the second computation was considerably more efficient. Similar results were observed for problem B2 of [4]. Ignoring discontinuities, there were 74 successful steps and 63 unsuccessful steps. Using the event function $g = y(t)$ and a derivative switching flag, there were 28 successful steps and no unsuccessful steps.

6 Form of the Solution

As already mentioned in §2, the default is to return the approximate solution at the mesh points selected by DDE_SOLVER. If the solution structure is called SOL, the number of mesh points NPTS is returned as SOL%NPTS, the mesh points $t_0 < t_1 < \ldots < t_f$ chosen by the solver are returned as SOL%T, and the corresponding approximate solutions are returned in the NPTS $\times$ NEQN array SOL%Y. The only difference when TSPAN has more than two entries is that SOL%T is then the same as TSPAN.

There are two other ways that the solver can return a solution. A general-purpose DDE solver must have a continuous extension that allows the solution to be approximated accurately between mesh points. The default output of `dde23/sd` is to return in the solution structure the information needed to evaluate these continuous extensions anywhere in the interval of integration. An auxiliary function `deval` is used for this purpose. In a major extension of DKLAG6, we have endowed DDE_SOLVER with this valuable capability. Because the capability requires a considerable amount of additional information in SOL, we have made this an option called INTERPOLATION. By default it has the value .FALSE. If DDE_SET is used to set this option to .TRUE., the solver includes in SOL the information needed for interpolation. A call to the auxiliary function DDE_VAL of the form

```
YINT = DDE_VAL(T,SOL)
```

evaluates approximations to $y(T)$ for any *vector* of points T that all lie in $[t_0, t_f]$. The values $y'(T)$ are also returned when an optional argument of DDE_VAL with keyword DERIVATIVES is set to .TRUE. Sometimes only selected components are of interest. The optional argument with keyword COMPONENTS is a vector that tells DDE_VAL which components of the solution are desired. For

example, the first and third components of the solution at the two arguments $t = 3.14, 1.41$ would be returned by

```
YINT = DDE_VAL((/ 3.14D0,1.41D0 /),SOL,COMPONENTS=(/ 1, 3 /))
```

The output argument is a derived type called DDE_INT, so YINT in this example must be declared as

```
TYPE(DDE_INT) :: YINT
```

Although the vector T is available in the calling program, it is convenient to return it as the field YINT%TVALS in the interpolation structure. Similarly, if a vector is provided for the keyword COMPONENTS, it is returned as the field YINT%COMPONENTS. If this option is not set, the field is given the default value of all the components, namely the vector with entries 1,. . . ,NEQN. The approximate solution is returned in the field YINT%YT. More specifically, YINT%YT(I,J) is an approximation to component J of the solution at T(I). Similarly, if DERIVATIVES=.TRUE., an approximation to component J of the first derivative of the solution at T(I) is returned in YINT%DT(I,J). In our first version of DDE_VAL, we used the binary search of DKLAG6, but we found that a smart linear search provided a considerable improvement in speed.

If the standard output options are not exactly what you want, you can write your own output function. For instance, if NEQN is large, you might find return of all components of the solution at all mesh points to be excessive. By means of an output function you could output only the components of particular interest. An output function must have the form

```
SUBROUTINE OUT(T,Y,DY,NEQN,NE)
```

The solver is informed of this using a keyword for this optional argument: OUT_FCN = OUT. The solver will call OUT at every step and every event. In this call T is the value of the independent variable and Y and DY are the NEQN components of the solution and its first derivative, respectively, at T. If NE is zero, the solver has stepped to T. If NE is positive, event function NE vanishes at T. You can inspect this information and do whatever you wish before returning control to the solver. For example, you might write a particular solution component to a file.

7 Neutral Problems

It is remarkably easy to *apply* DDE_SOLVER to a problem of neutral type—whether it will succeed in solving the problem is another matter. The first question is how to modify the coding (8) of the differential equations (4) to account for the values $y'(\beta_j(t, y(t))$. As noted in §2, the design of DKLAG6 uses another argument, DYLAG, in (9). The disadvantage of this design is that the argument must be present when solving the far more common retarded DDEs even though it is not used. To solve a neutral problem with DDE_SOLVER,

just set the option NEUTRAL = .TRUE. and extend the definition of Z in (8) so that the array is of size NEQN $\times$ 2 NLAGS. The first NLAGS columns correspond as usual to the $y(\beta_j(t, y(t)))$ and the next set of NLAGS columns correspond to $y'(\beta_j(t, y(t)))$. Similarly the definition of the array Y returned by the HISTORY function is extended so that the first column is still $y(t)$ and a second is now $y'(t)$. If the history is constant and supplied as a vector, the solver realizes that the derivative of the history is zero so that it is not necessary to add a column of zeros.

8 Advanced Delays

We require that the delay functions satisfy $\beta_j(t, y(t)) \leq t$, which is to say that no delayed argument can exceed the current time. However, for problems with small state-dependent delays, and especially problems with a delay that vanishes at some point in the course of the integration (singular problems), error in the computed solution can cause a delayed argument to exceed slightly the current time. For this reason DDE_SOLVER allows advanced delays. Because the continuous extension is determined iteratively as described in [14], an accurate solution can be computed in this situation. In §10 we present some numerical results for the solution of a singular problem of neutral type. Monitoring of that computation showed that there were 24 instances of advanced delays.

9 Discontinuity Tracking

A fundamental issue is whether to track discontinuities explicitly or to rely on the error control to handle them indirectly. In the latter approach it is hoped that the error estimator will recognize discontinuities and cause the solver to reduce the step size automatically to get past them without degrading the accuracy of the solution too much. The `ddesd` solver [10] takes a different approach. It does not track discontinuities, but it is intended only for problems with solutions that get smoother as the integration proceeds, hence not of neutral type, and it estimates and controls the residual in an attempt to handle discontinuities more robustly than conventional local error control. Other solvers that attempt to deal directly with discontinuities do so in different ways. Since it is limited by design to DDEs with constant lags, `dde23` [14] builds the discontinuity tree at the beginning of the integration and then steps exactly to each point in the tree (up to the order of the Runge-Kutta method used in the code) to avoid interpolating across a discontinuity. The situation is more troublesome when the delays are time- and state-dependent. Although none of the tracking methods in use is entirely satisfactory, they all work well enough in practice and improve both the efficiency and the reliability of the solvers. DDVERK [4] uses a method based on the behavior of the solver following error test failures. When a discontinuity is suspected, bisection is used to locate it and then special interpolants are used to get past it. The user of ARCHI [9] can provide a diagram that

defines the dependence of the DDEs on the various delays. The solver uses it to track the discontinuities explicitly, locating them precisely with a root finding subroutine. When all delays are constant, DDE_SOLVER builds the tree in a manner similar to `dde23` and steps to each point in the tree exactly, thereby eliminating the need for root finding as well as the necessity to use questionable error estimates near discontinuities. When the delays are state-dependent, DDE_SOLVER tracks discontinuities explicitly using root finding in the manner described below.

Because we believe that tracking discontinuities should be invisible to the user, DDE_SOLVER does not ask for any information about discontinuities, with one minor exception. We assume that there is a low-order discontinuity at the initial point which propagates because of the effects of delays. The exception is that sometimes there are low-order discontinuities in the history function, too. For example, one of the components of the solution of the Marchuk immunology problem solved as Exercise 4.8 of [11] has $\max(0, t + 10^{-6})$ as its history for $t \leq 0$. The discontinuity in the first derivative at $t = -10^{-6}$ propagates into the interval of integration. The history function is discontinuous at t_0 for Example 4.4.4 of [11]. The solver can be told about points of discontinuity at points known in advance by providing them in a vector as the value of the option JUMPS. The default is that there are no points of this kind. For the sake of clarity, we describe here only the most common situation of discontinuities propagating from t_0.

For each delay the solver starts with an event function $g = \beta_j(t, y(t)) - t_0$. Each time a root t^* is located, an event function $g = \beta_j(t, y(t)) - t^*$ is added. For non-neutral problems it is assumed that the solution is smoothed as the integration progresses, so this process is continued until the level of smoothing exceeds the order of the Runge-Kutta method. If the JUMPS option is used, the tracking is continued to one higher level because the solution itself might be discontinuous at such points. The Runge-Kutta method of DDE_SOLVER has an interpolant (continuous extension), meaning that it approximates a solution component not just at the end of a step, but throughout the span of a step by a polynomial. These polynomials are used in conjunction with a root-finding subroutine to locate discontinuities following each successful step. Though this means that the solver will sometimes interpolate near a discontinuity, special precautions are taken at such a point to minimize the effect of the discontinuity. By default DDE_SOLVER assumes that smoothing does not occur for neutral problems and so continues to build the discontinuity tree throughout the integration. If the problem is such that the effect of the derivative discontinuities lessens as the integration proceeds, it is possible to override this default and so improve considerably the efficiency of the solver.

The way that DDE_SOLVER handles discontinuities is generally quite satisfactory, but there are some options for changing this that are sometimes useful. By default the solver tracks all discontinuities for neutral problems and tracks them to seven levels for all other problems. You can disable the tracking of discontinuities entirely by setting the option TRACK_DISCONTINUITIES to .FALSE. You can change the maximum level of tracking by setting the option

PRINT_STATS	Subroutine to print statistics for the run
DDE_SET	Function used to set options
DDE_VAL	Function to evaluate the solution and its derivative

Table 4: Auxiliary procedures for DDE_SOLVER.

TRACKING_LEVEL to the maximum level you prefer. You can track all discontinuities by setting the option to zero.

10 Illustrative Computations

Chapter 4 of [11] has examples that represent a wide variety of problems for DDEs. Though most are solved with `dde23`, there is a section devoted to other kinds of DDEs and software that includes examples of singular and neutral DDEs solved with DKLAG6. We have solved all these examples with DDE_SOLVER. In most cases the numerical results were exported to MATLAB and plotted there. We have written M-files for this purpose. Besides showing how easy it is to solve even rather complicated tasks with DDE_SOLVER, the examples can serve as templates. DDE_SOLVER is accompanied by the four auxiliary procedures of Table 4 that were described in §2, §5.1, and §6. All these examples and the module for DDE_SOLVER are available at the web site [5].

In F90 it is possible to declare the INTENT of dummy arguments to procedures as being input, output, or both. It is not necessary to do this, but to make the roles of arguments as clear as possible, we have done so in the examples that follow.

Models of self-pulsing lasers with delayed feedback are formulated in T.W. Carr [1, 2]. Carr has been solving these DDEs in MATLAB with `dde23`, but some of the computations are lengthy. Although it is not as easy to code the solution of these models with DDE_SOLVER, it is not much harder. In compensation for extra work writing the program and exporting the solution for plotting, DDE_SOLVER is dramatically faster. `DKLAG6`, `dde23`, and `ddesd` are all based on explicit Runge-Kutta formulas. The first two are based on pairs of orders (5,6) and (2,3), respectively. The third estimates its error in a different way that is roughly equivalent to a (3,4) pair. The difference in order is significant for the tolerances of this example, but it is the difference between interpreted and compiled computation that accounts for most of reduction in run time. We illustrate the use of DDE_SOLVER with an example of a self-pulsing semiconductor laser with an external cavity [1]. The task is defined in a module that we reproduce here with all blank lines removed for brevity:

```
MODULE define_DDEs
  IMPLICIT NONE
  INTEGER, PARAMETER :: NEQN=3,NLAGS=1,NEF=1
  ! Physical parameters
```

```
  DOUBLE PRECISION :: A1,A2,a,g,alpha,omegan,eta
CONTAINS
  SUBROUTINE DDES(T,Y,Z,DYDT)
    DOUBLE PRECISION :: T
    DOUBLE PRECISION, DIMENSION(NEQN) :: Y,DYDT
    DOUBLE PRECISION, DIMENSION(NEQN,NLAGS) :: Z
    INTENT(IN)  :: T,Y,Z
    INTENT(OUT) :: DYDT
    ! Local variables
    DOUBLE PRECISION :: E,D,P,ylag1,ylag3
    E = Y(1)
    D = Y(2)
    P = Y(3)
    ylag1 = Z(1,1)       ! Retarded field.
    ylag3 = Z(3,1)       ! Retarded phase.
    IF (T < 200D0) THEN
      eta = 0D0
    ELSE
      eta = 0.2D0
    END IF
    DYDT(1) = (0.5D0)*(D + A2/(1D0 + a*E**2) - 1D0)*E &
                           + eta*ylag1*COS(ylag3 - P)
    DYDT(2) = g*(A1 - (1D0 + E**2)*D)
    DYDT(3) = omegan + (0.5D0*alpha)*(D + A2/(1D0 + a*E**2)) &
                             + eta*(ylag1/E)*SIN(ylag3 - P)
    RETURN
  END SUBROUTINE DDES
  SUBROUTINE EF(T,Y,DYDT,Z,G)
    DOUBLE PRECISION :: T
    DOUBLE PRECISION, DIMENSION(NEQN) :: Y,DYDT
    DOUBLE PRECISION, DIMENSION(NEQN,NLAGS) :: Z
    DOUBLE PRECISION, DIMENSION(NEF) :: G
    INTENT(IN)  :: T,Y,DYDT,Z
    INTENT(OUT) :: G
    G(1) = A1 - (1D0 + Y(1)**2)*Y(2)
    RETURN
  END SUBROUTINE EF
END MODULE define_DDEs
```

This module is USEed in the main program where the physical parameters are assigned values. For brevity we provide only portions of the program. The complete program, `laserex.f90`, and an M-file to plot the solution are available from the web site.

```
PROGRAM laserex
  .
```

```
  USE define_DDEs
  USE DDE_SOLVER_M
  .

  INTEGER, DIMENSION(3) :: NVAR=(/ NEQN,NLAGS,NEF /)
  TYPE(DDE_SOL) :: SOL
  TYPE(DDE_OPTS) :: OPTS
  DOUBLE PRECISION, DIMENSION(NEQN) :: HISTORY= &
     (/ 8.69863300579174D0, 0.11387116103411D0, 0D0 /)
  INTEGER :: I,J  ! Local variables
  DOUBLE PRECISION, PARAMETER :: DELAY=20D0,T0=0D0,TF=400D0

  ! For plotting y(t) against y(t - delay) in LASEREX.M,
  ! the mesh must have the following form.  Note that the
  ! last point of the integration,TSPAN(NOUT), may not be
  ! exactly TF.
  DOUBLE PRECISION, PARAMETER :: REFINE=50,SPACING=DELAY/REFINE
  INTEGER, PARAMETER :: NOUT = 1 + (TF - T0)/SPACING
  DOUBLE PRECISION, DIMENSION(NOUT) :: &
                   TSPAN = (/ (T0 + (I-1)*SPACING, I=1,NOUT) /)
  .

  OPTS = DDE_SET(RE=1D-5,AE=1D-9,DIRECTION=(/ -1 /),&
                 JUMPS=(/ 200D0 /))
  SOL = DDE_SOLVER(NVAR,DDES,(/ DELAY /),HISTORY,&
                   TSPAN,OPTIONS=OPTS,EVENT_FCN=EF)
  .

END PROGRAM laserex
```

The number of equations, NEQN, the number of delays, NLAGS, and the number of event functions, NEF, are defined in `define_DDEs` as PARAMETERs so that they can be used for dimensioning arrays in the program. The output SOL is declared to be of the DDE_SOL derived type. Options are set, so we similarly define a variable OPTS as being of type DDE_OPTS. The one delay function is $t - 20$, so we can define it by supplying the parameter DELAY in an array. The history function is constant, so we can define it by the array HISTORY. A variety of plots are to be made that involve components of $y(t)$ and $y(t - 20)$. Accordingly we define TSPAN as an array of NOUT equally spaced points. The parameter REFINE specifies how many output points are to be returned in each interval of length DELAY. It is used to determine NOUT and the SPACING, which are subsequently used in the formation of TSPAN with an array constructor and implied DO loop. Scalar tolerances more stringent than the defaults are set using DDE_SET as options in a structure OPTS and passed to the solver using the keyword OPTIONS. The name of the event function is passed to the solver using the keyword EVENT_FCN. This event function is

used to locate the local maxima of a quantity called "inversion" by finding where its first derivative vanishes. Minima are excluded by setting the DIRECTION option to -1. Notice that the parameter `eta` changes in the subroutine DDES from 0 to 0.2 at $t = 200$. Although the solver can cope with this on its own, it is better to use the JUMPS option to inform the solver about the discontinuity.

We do not show the portions of the program concerned with displaying information about the solution and events on the screen and writing them to files. The results were entirely consistent with those computed in MATLAB using `dde23`. We remark that there were 13 events.

To illustrate other features of DDE_SOLVER, we solve an artificial problem found in Chapter 4 of [11]. The differential equation

$$\begin{aligned} y'(t) &= \cos(t)[1 + y(ty^2(t))] + 0.3y(t)y'(ty^2(t)) \\ &\quad +0.7\sin(t)\cos(t\sin^2(t)) - \sin(t + t\sin^2(t)) \end{aligned} \tag{10}$$

is to be solved on $[0, \pi/2]$. This equation presents several difficulties: It is of neutral type, the one delay function $\beta(t, y(t)) = ty^2(t)$ is time- and state-dependent, and the delay is singular at $t = 0$ and again at $t = \frac{\pi}{2}$. With the history $\sin(t)$, the problem has the analytical solution $y(t) = \sin(t)$ that we can use to verify the accuracy of the numerical solution. This problem is an initial-value DDE, but no special provision is made for such problems in DDE_SOLVER. The subroutines defining the problem follow with blank lines removed for brevity, but some comments about neutral problems left in. Notice how we have extended arrays to accommodate the additional information required when solving a problem of neutral type. In particular, we have to provide the derivative of the history as well as the history itself.

```
SUBROUTINE DDES(T,Y,Z,DY)
  DOUBLE PRECISION :: T
  DOUBLE PRECISION, DIMENSION(NEQN) :: Y,DY,YLAG,DYLAG
  DOUBLE PRECISION, DIMENSION(NEQN,2*NLAGS) :: Z
  DOUBLE PRECISION, DIMENSION(NEQN,NLAGS) :: YLAG,DYLAG
  INTENT(IN)  :: T,Y,Z
  INTENT(OUT) :: DY
  ! NOTE:
  ! This is a neutral problem. Columns 1...NLAGS of the Z
  ! matrix contain the delayed solution values and columns
  ! NLAGS+1...2*NLAGS contain the delayed derivative values.
  YLAG  = Z(:,1:NLAGS)
  DYLAG = Z(:,NLAGS+1:2*NLAGS)
  DY(1) = COS(T)*(1D0 + YLAG(1,1)) + 0.3D0*Y(1)*DYLAG(1,1)  &
      + 0.7D0*SIN(T)*COS( T*SIN(T)**2 ) - SIN( T+T*SIN(T)**2 )
  RETURN
END SUBROUTINE DDES
SUBROUTINE HISTORY(T,Y)
  DOUBLE PRECISION :: T
```

```
  DOUBLE PRECISION, DIMENSION(NEQN,2) :: Y
  INTENT(IN)  :: T
  INTENT(OUT) :: Y
  ! NOTE:
  ! This is a neutral problem.  Return the history
  ! in column 1 of the Y matrix and the derivative
  ! of the history in column 2.
  Y(1,1) = SIN(T)
  Y(1,2) = COS(T)
  RETURN
END SUBROUTINE HISTORY
SUBROUTINE BETA(T,Y,BVAL)
  DOUBLE PRECISION :: T
  DOUBLE PRECISION, DIMENSION(NEQN) :: Y
  DOUBLE PRECISION, DIMENSION(NLAGS) :: BVAL
  INTENT(IN)  :: T,Y
  INTENT(OUT) :: BVAL
  BVAL(1) = T*Y(1)**2
  RETURN
END SUBROUTINE BETA
```

The complete program, `neutvan.f90`, and an M-file to plot the solution are available from the web site. The key lines of code in the main program are

```
OPTS = DDE_SET(RE=1D-10,AE=1D-10,NEUTRAL=.TRUE.)
SOL = DDE_SOLVER(NVAR,DDES,BETA,HISTORY,&
                 (/ 0D0,ASIN(1D0) /),OPTIONS=OPTS)
```

Using DDE_SET we told the solver that this is a neutral problem and set tolerances that are quite stringent. Even so, the computation was not expensive. By calling PRINT_STATS we obtained the integration statistics

```
Number of successful steps                  =   22
Number of failed steps                      =    4
Number of derivative evaluations            =  567
Number of failed corrector iterations       =    2
Total array storage used                    = 1156
Total number of root finding functions used =    1
```

Because TSPAN has two entries, the solution at all mesh points is returned in SOL. Using the analytical solution we determined the maximum absolute error at these points to be about 10^{-11}, a very satisfactory outcome.

11 Conclusions

We developed DDE_SOLVER from DKLAG6. The source code for DKLAG6 is distributed in a file of some 18,000 lines. The demonstrated effectiveness of

DKLAG6 and the size of this file explain why we wanted to start from the legacy code instead of writing an entirely new program. Like all the popular Fortran DDE solvers, new users find DKLAG6 formidable and in some respects, it is not easy to use. We have applied our experience writing DDE solvers in MATLAB and the capabilities of Fortran 90 to the development of a friendly Fortran DDE solver. How we accomplished this may be of interest to others who would like to modernize legacy code.

Although accomplishing everything we wanted to do resulted in significant restructuring of the legacy code, it was worth the effort since we believe DDE_SOLVER is superior by far to DKLAG6. A user has to provide much less information and what must be provided is more natural and can be coded in an easier way. In the course of developing a completely new user interface, we have added significantly to the capabilities of DKLAG6, capabilities that make DDE_SOLVER more comparable to `dde23/sd`. Although our original plan was to use the legacy code to the extent possible, we could not resist the temptation to eliminate large portions of the code and to exploit F90 to restructure the rest. The source code file for DDE_SOLVER is roughly one third the size of that for DKLAG6. This file and a collection of example programs are available from the web site [5].

`dde23` is capable of solving most constant lag problems of interest. Similarly, `ddesd` is capable of solving most time- and state-dependent delay problems. Although solving a given problem with DDE_SOLVER is not as easy as solving it with `dde23/sd`, it is not much harder and the difference is largely due to writing programs in F90 instead of MATLAB. In some circumstances DDE_SOLVER is clearly the best of these three DDE solvers: It is *much* faster than `dde23/sd`, so it should be used whenever run time is important. Finally, it can be applied to DDEs of neutral type while `dde23/sd` cannot.

12 Acknowledgment

We are grateful to T.W. Carr for providing the physical example of §10 and a MATLAB program for solving it with `dde23`. Our conversations about the effective solution of DDEs arising in [1, 2] influenced the design of DDE_SOLVER.

References

[1] T.W. Carr, Onset of instabilities in self-pulsing semiconductor lasers with delayed feedback, Euro. Phys. J. D., 19 (2002) 245–255.

[2] T.W. Carr, Period-locking due to delayed feedback in a laser with saturable absorber, Phys. Rev. E, 68:026212, 2003.

[3] S.P. Corwin, D. Sarafyan, and S. Thompson, DKLAG6: A code based on continuously imbedded sixth order Runge-Kutta methods for the solution

of state dependent functional differential equations, Appl. Num. Math., 24 (1997) 319–333.

[4] W.H. Enright and H. Hayashi, A Delay Differential Equation Solver Based on a Continuous Runge-Kutta Method with Defect Control, Numer. Alg. 16 (1997) 349–364.

[5] Friendly Fortran DDE Solver: http://www.radford.edu/∼thompson/ffddes/.

[6] C.L. Lawson, R.J. Hanson, F.T. Krogh, and O.R. Kincaid, Basic Linear Algebra Subprograms for FORTRAN usage, ACM Trans. Math. Softw., 5 (1979) 308–323.

[7] MATLAB 6, The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, 2002.

[8] K.W. Neves, Automatic integration of functional differential equations, ACM Trans. Math. Softw., 51 (1975) 357–368.

[9] C.A.H. Paul, A user-guide to ARCHI, Numer. Anal. Rept. No. 283, Maths. Dept., Univ. of Manchester, Manchester, UK, 1995.

[10] L.F. Shampine, Solving ODEs and DDEs with residual control, http://faculty.smu.edu/lshampin/residuals.pdf.

[11] L.F. Shampine, I. Gladwell, and S. Thompson, Solving ODEs with MATLAB, Cambridge Univ. Press, New York, 2003.

[12] L.F. Shampine and M.W. Reichelt, The MATLAB ODE Suite, SIAM J. Sci. Comput., 18 (1997) 1–22.

[13] L.F. Shampine and S. Thompson, Event location for ordinary differential equations, Comp. & Maths. with Appls., 39 (2000) 43–54.

[14] L.F. Shampine and S. Thompson, Solving DDEs in MATLAB, Appl. Numer. Math. 37 (2001) 441–458.

[15] L.F. Shampine and H.A. Watts, The art of writing a Runge-Kutta code, Part I, pp. 257-275 in J.R. Rice, ed., Mathematical Software III, Academic, New York, 1977; and The art of writing a Runge-Kutta code, II, Appl. Math. Comp. 5 (1979) 93–121.

[16] S. Suherman, R.H. Plaut, L.T. Watson, and S. Thompson, Effect of human response time on rocking instability of a two-wheeled suitcase, J. Sound and Vibration, 207 (1997) 617–625.