



A User-Guide to *Archi*
An Explicit Runge-Kutta Code for Solving
Delay and Neutral Differential Equations and
Parameter Estimation Problems

C.A.H. Paul

Numerical Analysis Report No. 283
(Extended)

April 1997

Manchester Centre for Computational Mathematics
Numerical Analysis Reports

DEPARTMENTS OF MATHEMATICS

Reports available from: And over the World-Wide Web from URLs
Department of Mathematics <http://www.ma.man.ac.uk/nareports>
University of Manchester <ftp://ftp.ma.man.ac.uk/pub/narep>
Manchester M13 9PL
England

A User-Guide to *Archi* – an Explicit Runge-Kutta Code for Solving Delay and Neutral Differential Equations and Parameter Estimation Problems

Christopher A.H. Paul

April 9, 1997

Version: 1.2

Background

The Fortran 77 code *Archi* was originally written as part of my Ph.D. in 1992¹, with the aim of solving delay differential equations (DDEs) and neutral differential equations (NDEs). The code was written with the intention of making it as portable as possible; thus the original *Archi* has undergone a number of minor modifications and improvements so as to maintain 100% compatibility with all the Fortran compilers available to me. Since 1992, two extended versions *Archi-L* (that interfaces with the NETLIB unconstrained optimization code LMDIF) and *Archi-N* (that interfaces with the NAG constrained optimization code E04UPF) for solving parameter estimation problems have been developed under SERC grant GR/H59237. The next extension to *Archi* (funded by EPSRC grant GR/K48297) is to add defect control, both as an error control for the solution and as an alternative to tracking derivative discontinuities (from now on referred to simply as ‘discontinuities’). This version of the guide now also documents the use of *Archi-L* and *Archi-N* for parameter estimation.

Archi is based on the fifth-order Dormand & Prince explicit Runge-Kutta method [5, p.23] with a fifth-order Hermite interpolant due to Shampine [12, p.149]. The choice of a Hermite interpolant was influenced by: (i) the conclusion of Gladwell, Shampine, Baca & Brankin [8, p.341] that for non-monotonic solutions, quintic Hermite interpolants are required in order to preserve the shape of a solution, and (ii) in order to maintain the asymptotic correctness of a p -th order error estimate, the local error in the approximation scheme used to evaluate delay terms must be at least order $p + 1$ [7, p.342]. *Archi* has also been designed to track the propagation of discontinuities in a solution using the discontinuity tracking theory developed by Willé & Baker [14], and to solve vanishing-lag DDEs using either extrapolation or the predictor-corrector approach developed by Baker & Paul [1, 2]. In addition, *Archi* can solve a limited class of delay-integro-differential equations, specifically those equations that have a sufficiently smooth integrand, using an adaptive quadrature algorithm developed by Paul [10].

This user-guide supercedes all other documentation on using *Archi*.

Introduction

There are three distinct stages to writing a driver program for solving a DDE or NDE: (i) Specifying the actual differential equation, (ii) specifying the discontinuity propagation network and related information, and (iii) altering *Archi*’s default values/behaviour.

Archi avoids the use of COMMON blocks by using two workspace arrays: INTGRS – an integer array, and REALS – a double precision array. *Archi* also uses a cyclic history queue, in which the oldest solution information is eventually over-written, in order to maximize the length of integration that can be solved.

¹Funded by the Science & Engineering Research Council and the Numerical Algorithms Group (Oxford) Ltd.

Contents

1	Writing a Main Program	4
1.1	Specifying the Range of Integration	4
1.2	Specifying the Initial Stepsize	4
1.3	Specifying the Local Error Tolerance	4
1.4	Solving the Delay Differential Equation	5
1.5	Changing the Dimension of the Problem	5
1.6	Changing the Number of Delay Functions	5
1.7	Specifying a Fixed Stepsize	5
1.8	Specifying a Maximum Stepsize	5
1.9	Changing the Level of Diagnostics	6
1.10	Printing Information about the Solution	6
1.10.1	Printing the Continuity Record	6
1.10.2	Printing the Numerical Derivative	6
1.10.3	Printing the Meshpoints	6
1.10.4	Printing the Numerical Solution	6
1.10.5	Locating Values of the Solution	7
1.11	Tracking Derivative Discontinuities	7
1.11.1	Changing the Level of Discontinuity Tracking	8
1.11.2	Specifying an Initial Continuity Record	8
1.11.3	Specifying a Strong Coupling Link	8
1.11.4	Specifying a Weak Coupling Link	8
1.11.5	Changing the Order upto which Discontinuities are Tracked	9
1.11.6	Changing the Upper Bound on the Lag Functions	9
1.11.7	Changing the Distance between Distinct Discontinuities	9
1.11.8	Changing the Length of the Continuity Record	9
1.11.9	Changing the Sizes of the Strong/Weak Coupling Networks	9
1.12	Vanishing-Lag Delay Differential Equations	9
1.12.1	Changing the Extrapolation Mode	10
1.12.2	Selecting the Predictor-Corrector Iteration	10
1.13	Solving Delay-Integro-Differential Equations	10
1.13.1	Changing the Number of Integral Delay Functions	10
1.13.2	Specifying the Smoothness of the Integrand	11
1.14	Changing the Level of Error Reporting	11
2	Writing a Derivative Function Routine	11
2.1	Evaluating a Solution Value	11
2.2	Evaluating a Derivative Value	11
2.3	Evaluating an Integral	12
3	Writing a Delay Function Routine	12
4	Writing an Initial Derivative/Solution Routine	12
5	Writing a Delay-Integro-Differential Routine	13
5.1	Specifying the Integral Delay Functions	13
6	<i>Archi's</i> Default Settings	14

7	Least-Squares Parameter Estimation using <i>Archi</i>	18
7.1	Specifying the Number of Parameters	18
7.2	Specifying the Parameter Tolerance	19
7.3	Specifying the Initial Point as a Parameter	19
7.4	Specifying an Initial Value as a Parameter	19
7.5	Specifying the Data to be Fitted	19
7.6	Solving the Minimization Problem	19
7.7	Printing the Results	19
7.7.1	Printing the Residual Vector	20
7.7.2	Printing the 2-Norm of the Residual Vector	20
7.8	Constrained Optimization Problems	20
7.8.1	Specifying the Number of Linear/Non-Linear Constraints	20
7.8.2	Specifying a Linear/Non-Linear Constraint Bound	20
7.8.3	Specifying a Bound on a Parameter	20
8	Writing a Linear/Non-Linear Constraint Routine	21

Throughout this report, I shall refer to the **mandatory skeleton** of a routine/function, this is the Fortran code that **must** appear in the routine/function.

Notation: Where a sample of Fortran code is given, quantities in **sans-serif** refer to INTEGERS, and quantities in *math-italic* refer to DOUBLE PRECISIONS.

1 Writing a Main Program

The workspace arrays and default values of *Archi* (which are summarized in §6) are initialized by:

```
CALL INIT(INTGRS,REALS,dimint,dimrel)
```

where `dimint` and `dimrel` are the dimensions of the integer and double precision workspace arrays, respectively. `INIT` also calculates the unit-roundoff μ .

The simplest driver program for a scalar DDE specifies the *range of integration*, the *initial stepsize*, the *local error tolerance* and the *initial values*. Its **mandatory skeleton** is:

```
PROGRAM ARCHI
DOUBLE PRECISION REALS(dimrel),Y(1)
INTEGER INTGRS(dimint)
CALL INIT(INTGRS,REALS,dimint,dimrel)
CALL RANGE(INTGRS,REALS,tSTART,tEND)
CALL HSTART(INTGRS,REALS,hSTART)
CALL ERROR(INTGRS,REALS,type,eps)
Y(1) = y0
CALL SOLVE(INTGRS,REALS,Y)
WRITE (*,*) Y
END
```

The **order of these routines** is important, because *Archi* includes a number of validation checks. For example, the range of integration must be given before the initial stepsize, because checks are made to ensure that the initial stepsize is not too large.

1.1 Specifying the Range of Integration

The *range of integration* is specified by:

```
CALL RANGE(INTGRS,REALS,tSTART,tEND)
```

The direction of integration must be positive, that is to say $t_{\text{START}} > t_{\text{END}}$, and once the range of integration has been specified it cannot be changed.

1.2 Specifying the Initial Stepsize

The *initial stepsize* h_{START} is specified by:

```
CALL HSTART(INTGRS,REALS,hSTART)
```

after the range of integration has been set (§1.1). An initial stepsize can only be set if a *fixed stepsize* h_{FIX} (§1.7) has not already been selected. h_{START} must be positive, not ‘too small’ ($h_{\text{START}} \geq 10\mu \max\{1, |t_{\text{START}}|\}$) and not ‘too large’ ($h_{\text{START}} \leq t_{\text{END}} - t_{\text{START}}$). Also, if a *maximum stepsize* h_{MAX} (§1.8) has been specified, then $h_{\text{START}} \leq h_{\text{MAX}}$.

1.3 Specifying the Local Error Tolerance

The *local error tolerance* and the *type of error estimator* are both specified by:

```
CALL ERROR(INTGRS,REALS,type,eps)
```

The *error-per-step tolerance* eps can only be set if a *fixed stepsize* h_{FIX} (§1.7) has not already been selected. The only constraint on eps is that $eps > \mu$. There are five different types of error estimate; the ‘standard’ embedded error estimate for the Dormand & Prince 5(4) method, and four ‘experimental’ error estimators. If $\hat{y}(t_n + \theta h_n)$ and $\tilde{y}(t_n + \theta h_n)$ are fourth- and fifth-order continuous extensions of the numerical solution, respectively, then for a scalar DDE

type	Definition of <i>eps</i>
1	$ \tilde{y}(t_n + h_n) - \hat{y}(t_n + h_n) $
2	$\int_{t_n}^{t_n+h_n} \tilde{y}(t) - \hat{y}(t) dt$
3	$\max_{t_n \leq t \leq t_n+h_n} \tilde{y}(t) - \hat{y}(t) h_n$
4	$\max_{\theta \in \{1/5, 3/10, 1/2, 4/5, 8/9, 1\}} \tilde{y}(t_n + \theta h_n) - \hat{y}(t_n + \theta h_n) h_n$
5	$\max_{\theta \in \{\xi: P_5(\xi)=0\}} \tilde{y}(t_n + \theta h_n) - \hat{y}(t_n + \theta h_n) h_n$

where $P_5(t)$ is the fifth-degree Chebyshev polynomial.

1.4 Solving the Delay Differential Equation

Once the initial values have been specified, the DDE is solved by:

CALL SOLVE(INTGRS, REALS, Y)

Upon successful completion, a self-explanatory list of diagnostics is given, although the solution at the end-point must be explicitly printed.

Archi can also solve systems of DDEs with multiple delay functions using either a fixed stepsize or a maximum stepsize, produce a variety of diagnostics, and present the results in a variety of ways.

1.5 Changing the Dimension of the Problem

The dimension of the system of DDEs can be changed from the default ($n = 1$) **once** by:

CALL DIMEN(INTGRS, REALS, dimension)

where $\text{dimension} > 0$.

1.6 Changing the Number of Delay Functions

The number of delay functions can be changed from the default ($n = 1$) **once** by:

CALL DELAYS(INTGRS, REALS, number)

where the number of delay functions must satisfy $\text{number} > 0$.

1.7 Specifying a Fixed Stepsize

Archi includes the facility to solve a DDE using a fixed stepsize. This is most useful when trying to establish the order of convergence of a solution, and how the order of convergence is affected by discontinuities. After the range of integration has been specified (§1.1), a fixed stepsize h_{FIX} is selected by:

CALL FIXSTP(INTGRS, REALS, h_{FIX})

where h_{FIX} must not be ‘too small’ ($h_{\text{FIX}} > 10\mu \max\{1, |t_{\text{START}}|\}$) nor ‘too large’ ($h_{\text{FIX}} \leq t_{\text{END}} - t_{\text{START}}$). If a fixed stepsize is selected, any attempt to specify an initial stepsize h_{START} (§1.2) or a maximum stepsize h_{MAX} (§1.8), to set a local error tolerance *eps* (§1.3), or to track discontinuities (§1.11.1) is ignored.

1.8 Specifying a Maximum Stepsize

In addition to specifying a local error tolerance (§1.3), *Archi* allows a maximum stepsize to be specified. After the range of integration has been specified (§1.1), a maximum stepsize h_{MAX} is set by:

CALL MAXSTP(INTGRS, REALS, h_{MAX})

where h_{MAX} must not be ‘too small’ ($h_{\text{MAX}} \geq 1000\mu \max\{1, |t_{\text{START}}|\}$) and not ‘too large’ ($h_{\text{MAX}} \leq t_{\text{END}} - t_{\text{START}}$). In addition, h_{MAX} is ignored if a fixed stepsize (§1.7) has been selected, and if h_{START} has already been specified then it must satisfy $h_{\text{START}} \leq h_{\text{MAX}}$.

1.9 Changing the Level of Diagnostics

By default, *Archi* produces a list of self-explanatory diagnostics after the successful integration of the DDE (level 1). However it is often useful if no diagnostics are produced, for example, when printing the numerical solution (§1.10.4) to a file for further processing. It is also useful, sometimes, to have limited diagnostics printed while the DDE is being solved, for example, to determine where most rejected steps occur. *Archi* allows the user to select four levels of diagnostics by:

```
CALL STATS(INTGRS,REALS,level)
```

where

level	Description of Diagnostics
0	No diagnostics printed
1	Diagnostics printed after successful integration
2	As 1 + number of steps, derivative calls, delay calls and extrapolations during run
3	As 1 + the current interval being solved during code execution

1.10 Printing Information about the Solution

There are five ways of printing information about the solution, in addition to printing the diagnostics (§1.9) and the solution at the end-point:

1.10.1 Printing the Continuity Record

If discontinuity tracking has been specified (§1.11.1), it is possible to print the continuity record (see page 15) to see how the discontinuities have propagated within and between solution components by:

```
CALL PRNTCO(INTGRS,REALS)
```

1.10.2 Printing the Numerical Derivative

It is possible to print the numerical derivative over an interval at uniform points by:

```
CALL PRNTDY(INTGRS,REALS,derivative,from,to,steps)
```

So long as the solution component `derivative` exists, $from < t_{END}$ and $steps > 0$, *Archi* will attempt to print out the numerical derivative over the interval specified. However, if the output point t_i is no longer in the history queue or $t_i > t_{END}$, an appropriate warning is issued.

1.10.3 Printing the Meshpoints

Having successfully solved a DDE, it is often useful to know where the meshpoints occur so that, for example, a graph of accepted stepsize against position can be produced. The meshpoints *in the history queue* can be printed by:

```
CALL PRNTTI(INTGRS,REALS)
```

1.10.4 Printing the Numerical Solution

There are two ways of printing the numerical solution: one is by calling the routine `PRNTY` to print the solution over an interval at uniform points, the other is by calling the double precision function `SOLNYI` that returns the value of the solution at the specified point.

The solution can be printed over an interval at uniform points by:

```
CALL PRNTY(INTGRS,REALS,solution,from,to,steps)
```

So long as the solution component `solution` exists, $from < t_{END}$ and $steps > 0$, *Archi* will attempt to print out the solution over the interval specified. However if the output point t_i is no longer in the history queue or $t_i > t_{END}$, an appropriate message is issued.

The value of the solution at a specified point can be found by calling the double precision function:

```
SOLNYI(INTGRS,REALS,solution,at)
```

So long as the solution component `solution` exists and `at` is in the history queue, `SOLNYI` returns the value of the solution at the point $t = at$ (otherwise it returns the value `ODO` and issues a warning).

1.10.5 Locating Values of the Solution

For a given solution value *value*, it is possible to determine the point t at which $y_{\text{solution}}(t) = \text{value}$ by calling the double precision function:

`YSTOP(INTGRS, REALS, solution, t0, value)`

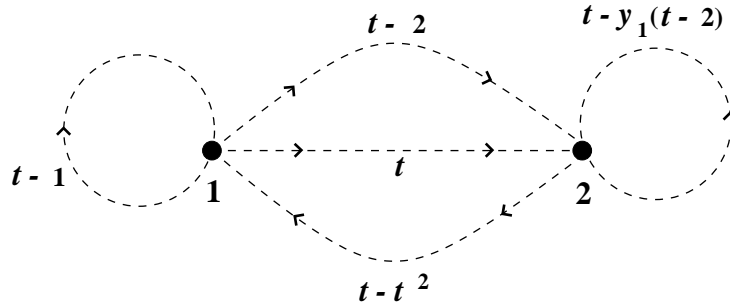
where `solution` must be a valid solution component and t_0 is the starting iterate. If the maximum number of iterations (50) is exceeded, or the iteration requires a solution value not in the history queue, then a warning is issued and the value `ODO` is returned.

1.11 Tracking Derivative Discontinuities

There are several routines associated with the tracking of discontinuities. First, however, it is necessary to review how the *network dependency graph* (see Willé & Baker [14]) is determined from a system of DDEs. Consider the two-dimensional system of DDEs

$$\begin{aligned} y_1'(t) &= y_1(t-1) + y_2(t-t^2), \\ y_2'(t) &= y_1(t) + y_2(t-y_1(t-2)). \end{aligned}$$

A discontinuity is propagated within the solution $y_1(t)$ when the delay function $\alpha_1(t) := t-1$ crosses the position of a discontinuity in $y_1(t)$. In addition, $y_1(t)$ inherits a discontinuity from $y_2(t)$ when $\alpha_2(t) := t-t^2$ crosses the position of a discontinuity in $y_2(t)$. The solution $y_2(t)$ inherits a discontinuity from $y_1(t)$ when $\alpha_3(t) := t$ crosses the position of a discontinuity in $y_1(t)$, and a discontinuity is propagated within $y_2(t)$ when $\alpha_4(t) := t-y_1(t-2)$ crosses the position of a discontinuity in $y_2(t)$. In addition, though, $y_2(t)$ inherits a further discontinuity from $y_1(t)$ when $\alpha_5(t) := t-2$ crosses the position of a discontinuity in $y_1(t)$ (due to $\alpha_4(t)$ being state-dependent). All this information may be represented graphically in the following *network dependency graph*:



Thus there are two distinct types of discontinuity propagation:

- Discontinuities that are ‘instantaneously’ propagated between different solution components (called *strong coupling* by Willé & Baker [14]).
- Discontinuities that are propagated ‘through time’ within the same solution component or between different solution components (called *weak coupling* by Willé & Baker).

It is *weak coupling* that makes the solution of differential equations with delayed terms challenging.

It is well-known that (for a DDE) as a discontinuity propagates, it is smoothed – that is to say, the discontinuity occurs in a higher derivative. Given that a discontinuity occurs at the point $t = \sigma_i$, it is propagated by the delay function $\alpha(t)$ to the point σ_j if $(\alpha(\sigma_j+) - \sigma_i) \times (\alpha(\sigma_j-) - \sigma_i) < 0$. This means that if $s = \sigma_j$ is an even-zero of

$$\alpha(s) = \sigma_i \tag{1}$$

no discontinuity is propagated, and if it is an odd multiple-zero of (1) then it is smoothed even more. However, it has been established that multiple-zeros of (1) are numerically ill-conditioned [6], and so it is usual to assume that all zeros of (1) are single zeros (with one degree of smoothing). (In the case of NDEs, there is typically no smoothing of a discontinuity as it is propagated.)

1.11.1 Changing the Level of Discontinuity Tracking

Archi allows the user to choose between not tracking discontinuities (the default), tracking only those discontinuities propagated by state-independent delay functions, and tracking all discontinuities. The level of tracking is changed by:

CALL TRACK (INTGRS, REALS, level)

so long as a fixed stepsize has not been selected (§1.7).

level	Description of Tracking
0	No tracking of discontinuities
1	Tracking discontinuities propagated by state-independent delay functions
2	Tracking all discontinuities

1.11.2 Specifying an Initial Continuity Record

In order to track discontinuities, it is necessary to specify the position and degree of smoothness of each discontinuity in the *initial function*, and at the initial point t_{START} . This is done for each discontinuity by:

CALL CONTY (INTGRS, REALS, component, smoothness, at)

where *component* is the solution component in which the discontinuity occurs, *smoothness* is the derivative in which the discontinuity occurs and *at* is the position of the discontinuity. So long as *component* is a valid solution component, *smoothness* ≥ 0 and there is room in the continuity record (§1.11.8), the discontinuity is recorded. **Note:** There is no check to ensure that $at \leq t_{\text{START}}$, so that it is possible to specify discontinuities beyond t_{START} , for example, in the case of a discontinuous derivative function

$$y'(t) = \begin{cases} y(t-1) & (0 \leq t \leq 5), \\ y(t-2) & (t > 5). \end{cases}$$

1.11.3 Specifying a Strong Coupling Link

Strong coupling is related to the instantaneous propagation of discontinuities between different solution components. Therefore, in order to specify the strong coupling in a system of DDEs, it is sufficient to specify the solution component in which the discontinuity occurs and the solution component into which it is propagated. Depending on whether the discontinuity is propagated by an ODE term $y_i(t)$ and is smoothed, or by a neutral term $y'_i(t)$ and is unsmoothed, determines how the strong coupling is specified. Strong coupling by an ODE term is specified by:

CALL HARD (INTGRS, REALS, from, to)

and for a neutral term by:

CALL NHARD (INTGRS, REALS, from, to)

So long as the length of the strong coupling network has not been reached (§1.11.9) and both solution components are valid, the link is added to the network.

1.11.4 Specifying a Weak Coupling Link

For weak coupling, in addition to specifying between which solution components a discontinuity is propagated, it is also necessary to specify which delay function propagates the discontinuity. As in the case of strong coupling (§1.11.3), the precise specification of the weak coupling is determined by whether the discontinuity is propagated by a delay term $y_i(t - \tau)$ or a neutral term $y'_i(t - \tau)$. Weak coupling by a delay term is specified by:

CALL SOFT (INTGRS, REALS, from, to, delayfn)

and for a neutral term by:

CALL NSOFT (INTGRS, REALS, from, to, delayfn)

So long as the size of the weak coupling network has not been reached (§1.11.9) and both the solution components and the delay function are valid, the link is added to the network.

1.11.5 Changing the Order upto which Discontinuities are Tracked

Although it is possible to track discontinuities upto any degree of smoothness, so long as the continuity record is long enough (§1.11.8), it is only really necessary to track discontinuities in derivatives upto the ‘natural’ order of the Runge-Kutta DDE method. However, it is possible to change this default limit ($n = 5$) by:

```
CALL CUTOFF(INTGRS,REALS,limit)
```

where $limit \geq 0$.

1.11.6 Changing the Upper Bound on the Lag Functions

One way that the computational cost of tracking discontinuities can be reduced is by specifying a bound τ_{upper} on the size of the lag functions $\{\tau_i(t)\}$. In doing so, all discontinuities that are older than τ_{upper} are no longer tracked, and so the overall cost of tracking is significantly reduced for small values of τ_{upper} . The default value of τ_{upper} is 10^{30} , and this can be changed by:

```
CALL LBOUND(INTGRS,REALS,bound)
```

where $bound \geq 0$.

1.11.7 Changing the Distance between Distinct Discontinuities

In order to track discontinuities, it is necessary to solve equations of the form (1) for $s = \sigma_j$. However, the floating-point version of the ‘tracking equation’ (1) is usually of the form

$$|\alpha(s) - \sigma_i| < \epsilon, \quad (2)$$

for sufficiently small $\epsilon \approx \sigma_i \mu$. Thus it is possible for a solution of (2) not to be a solution of (1), and for two solutions of (2), σ_j and σ_{j+1} say, to be “the same” ($|\sigma_{j+1} - \sigma_j| < \epsilon$). In order to avoid ‘duplicate’ discontinuities and some ‘spurious’ discontinuities (see, for example, (3)), it is possible to specify a minimum distance between discontinuities in the *same* solution component before they are considered to be distinct. The default distance (0.001) is changed by:

```
CALL SPACE(INTGRS,REALS,distance)
```

where $distance \geq 0$.

1.11.8 Changing the Length of the Continuity Record

The size of the two workspace arrays INTGRS and REALS is restricted by the amount of memory available. Thus it is necessary to balance the length of the history queue with the length of the continuity record. The default length of the continuity record ($n = 200$) can be changed by:

```
CALL STORCY(INTGRS,REALS,length)
```

before any continuity records (§1.11.2) and any strong or weak coupling links (§1.11.3, §1.11.4) have been specified. The minimum length of continuity record allowed is 10.

1.11.9 Changing the Sizes of the Strong/Weak Coupling Networks

The default sizes of the strong and weak coupling networks ($n = 25$) can be changed by:

```
CALL STORHD(INTGRS,REALS,length)
```

and

```
CALL STORST(INTGRS,REALS,length)
```

respectively. However, they can only be changed before any continuity records (§1.11.2) and any strong or weak coupling links (§1.11.3, §1.11.4) have been specified.

1.12 Vanishing-Lag Delay Differential Equations

For a vanishing-lag, $\alpha(t) \rightarrow t^*$ as $t \rightarrow t^*$ for some $t^* \geq t_{START}$. The most common example is that of an *initial value delay differential equation* (IVDDE), in which $\alpha(t_{START}) = t_{START}$. For example, the IVDDE $y'(t) = y(t^2)$ for $0 \leq t \leq 1$.

Archi offers the user two approaches to solving vanishing-lag DDEs: The first is the predictor-corrector (P-C) approach developed by Baker & Paul [1, 2] that can be used for any type of equation, and the second is extrapolation (that cannot be used for IVDDEs). *Archi* has three extrapolation modes: (i) no extrapolation, (ii) extrapolation within the current interval, and (iii) extrapolation beyond the current interval (which is useful when solving state-dependent problems that are “numerically advanced”).

1.12.1 Changing the Extrapolation Mode

The default extrapolation mode (mode 2) can be changed by:

```
CALL EXTRAP(INTGRS,REALS,mode)
```

where

mode	Description of Mode
1	No extrapolation
2	Extrapolation within the current interval only
3	Extrapolation beyond the current interval

If the DDE is an IVDDE and extrapolation has been specified (mode $\neq 1$), then the first interval is solved using the P-C iteration, but thereafter extrapolation is used.

1.12.2 Selecting the Predictor-Corrector Iteration

An alternative to using extrapolation for solving vanishing-lag DDEs is the P-C iteration of Baker & Paul (referred to as “iterative refinement” in *Archi*). The iteration is only performed when a delayed value lies within the current interval; if it lies beyond the current interval then an “advanced delay” error is reported. The iteration is considerably more expensive than the normal Runge-Kutta method, although there is considerable scope for parallelism. In order for the iteration to converge, it is necessary to track discontinuities (§1.11.1) upto the order of the DDE method ($n = 5$). The P-C iteration is selected by:

```
CALL REFINE(INTGRS,REALS)
```

1.13 Solving Delay-Integro-Differential Equations

Archi has the facility to solve a limited class of delay-integro-differential equations, specifically those in which the integrand is sufficiently smooth. The quadrature algorithm used is that developed by Paul [10], as it only requires a small fixed amount of storage (and is also reasonably fast).

1.13.1 Changing the Number of Integral Delay Functions

For efficiency, when evaluating delay functions, the delay functions corresponding to arguments in the integrand are specified separately. When the P-C iteration (§1.12.2) is used for vanishing-lag equations, the integration procedure can be (optionally) “accelerated” by only recalculating that part of the integral that can change during the iteration. The default number of “accelerated” integral terms ($n = 1$) and the default number of integral delay functions ($n = 1$) can be changed by:

```
CALL VOLTER(INTGRS,REALS,accelerate,number)
```

where `accelerate` ≥ 0 and `number` ≥ 0 , before any continuity records (§1.11.2) and any strong or weak coupling links (§1.11.3, §1.11.4) have been specified.

Note: Integral terms that have a state-dependent limit(s) of integration cannot be accelerated.

1.13.2 Specifying the Smoothness of the Integrand

By default, *Archi* uses a seventh-order Gauss-Legendre quadrature rule to approximate integral terms. However, if an integrand is likely to have poor continuity, for example, due to the presence of neutral terms or solution terms with large jumps in their low-order derivatives, then the efficiency of the adaptive quadrature can be severely effected. The efficiency of the adaptive quadrature may be significantly improved by

```
CALL VJUMP(INTGRS,REALS)
```

which specifies that *Archi* should use a fifth-order Gauss-Legendre quadrature rule for approximating integral terms.

1.14 Changing the Level of Error Reporting

Archi typically performs a number of checks on the data that is passed to it, and includes a simple information/warning/error message facility. How errors are treated can be changed from the default (level 3) by:

```
CALL REPORT(INTGRS,REALS,level)
```

where

level	Description of Action
0	No messages, automatic error correction where possible.
1	Report vanishing-lags, otherwise automatic error correction where possible.
2	Report all messages, but automatic error correction where possible.
3	Report all messages and halt code execution.

2 Writing a Derivative Function Routine

The derivative function is constructed using calls to the appropriate double precision functions which return either a solution value (§2.1), a derivative value (§2.2) or an integral value (§2.3). The **mandatory skeleton** of the derivative function is:

```
SUBROUTINE DERIV(INTGRS,REALS,T,F)
DOUBLE PRECISION DELAY,F(*),NEUTRL,REALS(*),T,VOLTRA
INTEGER INTGRS(*)
F(1) = .....
.....
F(dimension) = .....
RETURN
END
```

2.1 Evaluating a Solution Value

All solution values, including ODE terms $y_i(t)$ and values of the initial function (§4), are evaluated by calling the double precision function:

```
DELAY(INTGRS,REALS,lagfn,solution)
```

where *lagfn* must be a valid delay function (§3) and *solution* must be a valid solution component.

2.2 Evaluating a Derivative Value

All derivative values, including values of the initial derivative function (§4), are evaluated by calling the double precision function:

```
NEUTRL(INTGRS,REALS,lagfn,derivative)
```

where *lagfn* must be a valid delay function (§3) and *derivative* must be a valid solution component.

2.3 Evaluating an Integral

An integral term with limits given by two delay functions and an integrand given in a double precision function (§5) can be evaluated by calling the double precision function:

```
VOLTRA (INTGRS, REALS, T, lower, upper, kernel)
```

where `lower` and `upper` are the delay functions corresponding to the lower and upper limits of integration, respectively, and `kernel` corresponds to the `kernel`-th integrand in the `kernel` function (§5).

3 Writing a Delay Function Routine

The delay functions are specified in a double precision function `LAG`, with the appropriate delay function being selected using a computed-GOTO statement. The **mandatory skeleton** is:

```
DOUBLE PRECISION FUNCTION LAG (INTGRS, REALS, LAGFN, T)
DOUBLE PRECISION DELAY, NEUTRL, REALS (*), T
INTEGER INTGRS (*), LAGFN
GOTO (1, ...) LAGFN
1 LAG = .....
RETURN
.....
RETURN
END
```

where `LAGFN` is the delay function to be evaluated and `T` is the current point. There is a special built-in delay function, the *zeroth delay function*, that corresponds to the current point – equivalent to an ODE term. Thus the derivative function (§2) for the ODE $y'(t) = y(t)$ would be given as:

```
F(component) = DELAY (INTGRS, REALS, 0, solution)
```

The specification of state-dependent delay functions is just as straightforward. For example, the delay function $\alpha(t) = t - y_1(t)$ would be given as:

```
LAG = T-DELAY (INTGRS, REALS, 0, 1)
```

and neutral delay functions, such as $\alpha(t) = t - y_1'(t)$, can also be used. When discontinuities are being tracked, the order in which the delay functions are given is important for efficiency. Tracking discontinuities requires the delay functions to be evaluated repeatedly. Without knowing if the delay in a state-dependent delay function depends on other delay functions, such as $\alpha(t) = y(t^2)$, all the preceding delay functions must be evaluated. Thus state-dependent delay functions should be specified first, but not before their dependent delay functions. For example, $\alpha(t) = y_2(t^2)$ would be given as:

```
.....
1 LAG = T*T
RETURN
2 LAG = DELAY (INTGRS, REALS, 1, 2)
RETURN
.....
```

4 Writing an Initial Derivative/Solution Routine

Both the initial solution and the initial derivative are specified using double precision functions – `YINIT` and `FINIT`, respectively. Apart from the different function names, they are specified in exactly the same manner. The **mandatory skeleton** is:

```
DOUBLE PRECISION FUNCTION YINIT (SOLN, T)
DOUBLE PRECISION T
INTEGER SOLN
GOTO (1, ...) SOLN
1 YINIT = .....
RETURN
.....
RETURN
END
```

where the appropriate component of the function is selected using a computed-GOTO statement.

Note: Although the initial derivative should be the derivative of the initial solution, there are no checks to ensure that this is the case.

5 Writing a Delay-Integro-Differential Routine

The specification of this routine follows that of the initial derivative/solution, in that it is a double precision function and the appropriate integrand is selected using a computed-GOTO statement. The **mandatory skeleton** is:

```

DOUBLE PRECISION FUNCTION KERNEL(INTGRS,REALS,S,T,KERNFN)
DOUBLE PRECISION DELAY,NEUTRL,REALS(*),S,T
INTEGER INTGRS(*),KERNFN
GOTO (1,...) KERNFN
1 KERNEL = .....
RETURN
.....
RETURN
END

```

where the dependent and independent variables are S and T, respectively.

5.1 Specifying the Integral Delay Functions

There is a significant difference between the delay functions (§3) and the integral delay functions, in that the integral delay functions can depend on two variables – S and T. Therefore they cannot be specified with the delay functions (§3), although the same double precision functions DELAY and NEUTRL are used to evaluate delay and neutral terms. This difficulty is overcome by starting the numbering of integral delay functions from number + 1 (see §1.6). An integral delay function is evaluated by:

```
CALL VLAG(INTGRS,REALS,lagfn,lag)
```

where *lag* is the value of the integral delay function. For example, the integrand $y_3(s - t)$ can be evaluated using:

```

CALL VLAG(INTGRS,REALS,lagfn,S-T)
KERNEL = DELAY(INTGRS,REALS,lagfn,3)

```

As with DDEs and NDEs, discontinuities can propagate from integral terms into the solution [13]. A discontinuity is propagated whenever the lower or upper limit of integration crosses a discontinuity in the integrand. For example, a discontinuity is propagated into the solution by the term

$$\int_{t-2}^{t-1} y(s^2) ds$$

whenever $(t - 2)^2$ or $(t - 1)^2$ crosses a discontinuity in $y(t)$. Since the integral delay functions must be specified separately, in order to track these discontinuities it is necessary to specify the composite delay function in LAG (§3).

6 *Archi's* Default Settings

Description	Value	Routine
Extrapolation mode	Current interval only	EXTRAP/REFINE
Level of error handling	Halt code execution	REPORT
Level of diagnostics	After run completed	STATS
Level of discontinuity tracking	No tracking	TRACK
Number of delay functions	1	DELAYS
Dimension of DDE system	1	DIMEN
Distance between distinct discontinuities	0.001	SPACE
Upper bound on size of lag functions	10^{30}	LBOUND
Length of continuity record	200	STORCY
Size of strong coupling network	25	STORHD
Size of weak coupling network	25	STORST
Number of integral delays functions	1	VOLTER
Number of accelerated integral terms	1	VOLTER
Smoothness of integrand	Smooth	VJUMP

Sample Driver Programs and Results

The purpose of these examples is to illustrate how simple driver routines are written and to provide some comparison results to ensure that *Archi* is working correctly. (For references purposes, all results were obtained using `f77` on a Sun SPARC 10 running Solaris 5.4.)

The first example illustrates how to specify the discontinuity dependency network for a 2D system of DDEs with constant lags:

$$\begin{aligned} y_1'(t) &= y_1(t-1)y_2(t-2) & \Psi_1(t) &= \cos(t) & y_1(0) &= 0, \\ y_2'(t) &= -y_2(t-2)y_1(t) & \Psi_2(t) &= \sin(t) & y_2(0) &= 0, \end{aligned}$$

so that $y_1(t)$ has a jump at t_{START} and $y_2(t)$ is at least continuous at t_{START} .

```

PROGRAM ARCHI
DOUBLE PRECISION REALS(5000),Y(2)
INTEGER INTGRS(2000)
CALL INIT(INTGRS,REALS,5000,2000)           Specify size of workspace arrays
CALL DIMEN(INTGRS,REALS,2)                  Specify a 2D system of DDEs
CALL DELAYS(INTGRS,REALS,2)                Specify 2 delay functions
CALL RANGE(INTGRS,REALS,0D0,4D0)
CALL HSTART(INTGRS,REALS,1D0)
CALL ERROR(INTGRS,REALS,1,1D-9)            Embedded error estimator
CALL TRACK(INTGRS,REALS,2)                 Track all discontinuities
CALL CONTY(INTGRS,REALS,1,0,0D0)           Jump in y1(t) at t=0
CALL CONTY(INTGRS,REALS,2,1,0D0)           Jump in y2'(t) at t=0
CALL SOFT(INTGRS,REALS,1,1,1)              Weak link y1(t-1) -> y1'(t)
CALL SOFT(INTGRS,REALS,2,1,2)              Weak link y2(t-2) -> y1'(t)
CALL SOFT(INTGRS,REALS,2,2,2)              Weak link y2(t-2) -> y2'(t)
CALL HARD(INTGRS,REALS,1,2)                Strong link y1(t) -> y2'(t)
Y(1) = 0D0
Y(2) = 0D0
CALL SOLVE(INTGRS,REALS,Y)
CALL PRNTCO(INTGRS,REALS)                  Print continuity record
END

SUBROUTINE DERIV(INTGRS,REALS,T,F)
DOUBLE PRECISION DELAY,F(*),NEUTRL,REALS(*),T,VOLTRA
INTEGER INTGRS(*)
F(1) = DELAY(INTGRS,REALS,1,1)*DELAY(INTGRS,REALS,2,2)
F(2) = -DELAY(INTGRS,REALS,2,2)*DELAY(INTGRS,REALS,0,1)
RETURN
END

DOUBLE PRECISION FUNCTION LAG(INTGRS,REALS,LAGFN,T)

```

```

DOUBLE PRECISION DELAY,NEUTRL,REALS(*),T
INTEGER INTGRS(*),LAGFN
GOTO (1,2) LAGFN
1 LAG = T-1D0
RETURN
2 LAG = T-2D0
RETURN
END

DOUBLE PRECISION FUNCTION YINIT(SOLN,T)
DOUBLE PRECISION T
INTEGER SOLN
GOTO (1,2) SOLN
1 YINIT = COS(T)
RETURN
2 YINIT = SIN(T)
RETURN
END

```

Some Fortran compilers may also require `FINIT` and `KERNEL` to be given, even though the DDE does not actually use them. Also, the ANSI standard requires that all *Archi*'s functions and subroutines are specified in `EXTERNAL` statements. The results produced by `driver1.f` are:

```

Tracking all discontinuities
Length of cyclic storage is 134

Total number of steps      = 68
Number of accepted steps   = 62
Number of rejected steps   = 2
Number of derivative calls = 548
Number of approximant calls = 2192
Number of kernel calls     = 0
Number of extrapolations   = 0
Number of continuity records = 10

Maximum accepted stepsize  = 0.14313496026652
Minimum accepted stepsize  = 8.3255235493607D-03
Maximum extrapolation      = 0.H

```

Component	Smoothness	Position
1	0	0.
2	1	0.
1	1	1.0000000000000000
1	2	2.0000000000000000
2	2	2.0000000000000000
2	2	1.0000000000000000
1	3	3.0000000000000000
2	3	3.0000000000000000
1	3	4.0000000000000000
2	3	4.0000000000000000

The first thing to note is that the length of the history queue is 134. Its length is determined by the size of the two workspace arrays, and can typically be increased by increasing the size of the `REALS` array. (Increasing the size of the `INTGRS` array above 2000 usually has little effect.)

The number of ‘rejected initial steps’ (due to a too large a value of h_{START}) can be determined by subtracting the number of accepted and rejected steps from the total number of steps. Thus in the example above, there are $68 - (62 + 2) = 4$ rejected initial steps.

The next example illustrates how solutions of the floating-point tracking equation (2) can give rise to spurious discontinuities, that is to say, they are not solutions of the tracking equation (1). Consider the DDE

$$y'(t) = y(t - \frac{1}{10})y(t^2), \quad \Psi(t) = 1, \quad y(0) = -1, \quad (3)$$

that has a cluster-point at $t = 1$. Also note that, although (3) is not an IVDDE as it needs an initial function, $\alpha_2(t) := t^2$ has vanishing-lags at $t = t_{\text{START}}$ and $t = 1$. Thus the first step is

solved using the P-C iteration (§1.12.2), but as $t \rightarrow 1$ extrapolation (§1.12.1) is used to evaluate the $y(t^2)$ term.

```

PROGRAM ARCHI
DOUBLE PRECISION REALS(5000),Y(1)
INTEGER INTGRS(2000)
CALL INIT(INTGRS,REALS,5000,2000)
CALL REPORT(INTGRS,REALS,0)           Suppress all messages
CALL DELAYS(INTGRS,REALS,2)          Specify 2 delay functions
CALL RANGE(INTGRS,REALS,ODO,1DO)
CALL HSTART(INTGRS,REALS,0.1DO)
CALL ERROR(INTGRS,REALS,1,1D-6)      Embedded error estimator
CALL TRACK(INTGRS,REALS,2)           Track all discontinuities
CALL CONTY(INTGRS,REALS,1,0,ODO)     Jump in y(t) at t=0
CALL SOFT(INTGRS,REALS,1,1,1)        Weak link y(t-1/10) -> y'(t)
CALL SOFT(INTGRS,REALS,1,1,2)        Weak link y(t*t) -> y'(t)
Y(1) = -1DO
CALL SOLVE(INTGRS,REALS,Y)
WRITE (*,*) Y                         Print solution at endpoint
END

```

where $\alpha_1(t) := t - \frac{1}{10}$ and $\alpha_2(t) := t^2$. The results produced by `driver2.f` are:

Statistics for SPACE = 0.001		Statistics for SPACE = 0	
Total number of steps	= 30	Total number of steps	= 33
Number of accepted steps	= 29	Number of accepted steps	= 32
Number of rejected steps	= 1	Number of rejected steps	= 1
Number of derivative calls	= 301	Number of derivative calls	= 328
Number of approximant calls	= 602	Number of approximant calls	= 656
Number of kernel calls	= 0	Number of kernel calls	= 0
Number of extrapolations	= 23	Number of extrapolations	= 23
Number of continuity records	= 29	Number of continuity records	= 32
Maximum accepted stepsize	= 0.13403567663993	Maximum accepted stepsize	= 0.13403567663993
Minimum accepted stepsize	= 2.0564316888149D-03	Minimum accepted stepsize	= 3.4393468756966D-04
Maximum extrapolation	= 8.3406798738112H	Maximum extrapolation	= 8.3406798738112H
	-0.48389603579631		-0.48389603579527

The default ‘distance between distinct discontinuities’ (0.001) produces the results on the left. However, if this distance is reduced to zero by calling `SPACE` (§1.11.7), there is an increase in the number of discontinuities tracked. This increase is more significant when the P-C iteration is used instead of extrapolation (*see below*).

Another significant diagnostic is the ‘maximum (accepted) extrapolation’. Shampine has stated that the length of extrapolation in ODE codes should be restricted to $0.25H$ [11, p.1026], that is to say, for a continuous extension $y(t_n + \theta h_n)$, $0 \leq \theta \leq 1.25$. Clearly a value of $\theta = 9.3$ raises questions about the accuracy of the final solution (and the reliability of the extrapolation). This is one reason why the P-C iteration (§1.12.2) is included in *Archi*. If (3) is resolved using the P-C iteration to solve the DDE when the lag vanishes, then `driver2.f` produces the following results:

Statistics for SPACE = 0.001		Statistics for SPACE = 0	
Total number of steps	= 30	Total number of steps	= 47
Number of accepted steps	= 29	Number of accepted steps	= 46
Number of rejected steps	= 1	Number of rejected steps	= 1
Number of derivative calls	= 370	Number of derivative calls	= 523
Number of approximant calls	= 740	Number of approximant calls	= 1046
Number of kernel calls	= 0	Number of kernel calls	= 0
Number of extrapolations	= 0	Number of extrapolations	= 0
Number of continuity records	= 29	Number of continuity records	= 46
Maximum accepted stepsize	= 0.13403567663993	Maximum accepted stepsize	= 0.13403567663993
Minimum accepted stepsize	= 2.0564316888153D-03	Minimum accepted stepsize	= 7.7715611723761D-16
Maximum extrapolation	= 0.H	Maximum extrapolation	= 0.H

It is obvious from the statistics that there has been an increase in the number of derivative evaluations per step, from 10.0 using extrapolation to 12.3 using the P-C iteration. However, the important question is not how much more expensive is the P-C iteration compared to extrapolation, but how much more accurate/robust is it? A reference solution for (3) is -0.4838960765578336 , so that the relative error in the solutions is 8.4×10^{-8} and 1.9×10^{-8} , respectively.

The final example shows how to specify a delay-integro-differential equation, and how the P-C iteration can sometimes be considerably more efficient than extrapolation. The equation is

$$y'(t) = \frac{1}{t^3} \left(\int_{ty(t)}^{t^2 y(t)} s^3 y(s) y'(s) ds - 1 \right), \quad y(1) = 1,$$

which has the solution $y(t) = 1/t$.

```

PROGRAM ARCHI
DOUBLE PRECISION REALS(5000),Y(1)
INTEGER INTGRS(1000)
CALL INIT(INTGRS,REALS,1000,5000)
CALL REPORT(INTGRS,REALS,0)
CALL DELAYS(INTGRS,REALS,2)
CALL RANGE(INTGRS,REALS,1D0,2D0)
CALL HSTART(INTGRS,REALS,0.1D0)
CALL ERROR(INTGRS,REALS,1,1D-9)
Y(1) = 1D0
CALL SOLVE(INTGRS,REALS,Y)
END

SUBROUTINE DERIV(INTGRS,REALS,T,F)
DOUBLE PRECISION DELAY,F(*),NEUTRL,REALS(*),T,VOLTRA
INTEGER INTGRS(*)
F(1) = (VOLTRA(INTGRS,REALS,T,1,2,1)-1D0)/(T*T*T)
RETURN
END

DOUBLE PRECISION FUNCTION LAG(INTGRS,REALS,LAGFN,T)
DOUBLE PRECISION DELAY,NEUTRL,REALS(*),T
INTEGER INTGRS(*),LAGFN
GOTO (1,2) LAGFN
1 LAG = T*DELAY(INTGRS,REALS,0,1)
RETURN
2 LAG = T*T*DELAY(INTGRS,REALS,0,1)
RETURN
END

DOUBLE PRECISION FUNCTION YINIT(SOLN,T)
DOUBLE PRECISION T
INTEGER SOLN
YINIT = 1D0
RETURN
END

DOUBLE PRECISION FUNCTION FINIT(SOLN,T)
DOUBLE PRECISION T
INTEGER SOLN
FINIT = 0D0
RETURN
END

DOUBLE PRECISION FUNCTION KERNEL(INTGRS,REALS,S,T,KERNFN)
DOUBLE PRECISION DELAY,NEUTRL,REALS(*),S,T
INTEGER INTGRS(*),KERNFN
CALL VLAG(INTGRS,REALS,3,S)
KERNEL = S*S*S*DELAY(INTGRS,REALS,3,1)*NEUTRL(INTGRS,REALS,3,1)
RETURN
END

```

The results produced by `driver3.f` are:

Statistics for extrapolation		Statistics for P-C iteration	
Total number of steps	= 49	Total number of steps	= 16
Number of accepted steps	= 46	Number of accepted steps	= 15
Number of rejected steps	= 2	Number of rejected steps	= 0
Number of derivative calls	= 459	Number of derivative calls	= 657
Number of approximant calls	= 994330	Number of approximant calls	= 166604
Number of kernel calls	= 496819	Number of kernel calls	= 82804
Number of extrapolations	= 4500	Number of extrapolations	= 0
Number of continuity records	= 0	Number of continuity records	= 0
Maximum accepted stepsize	= 5.4387656204682D-02	Maximum accepted stepsize	= 9.0401181210989D-02
Minimum accepted stepsize	= 1.5165148759671D-02	Minimum accepted stepsize	= 5.2372075860289D-02
Maximum extrapolation	= 1.0131051676600H	Maximum extrapolation	= 0.H

Not only does the P-C iteration require fewer steps than extrapolation, the error in the solution is also significantly smaller.

7 Least-Squares Parameter Estimation using *Archi*

Parameter estimation in DDEs may be viewed as being the opposite of solving DDEs: Instead of being given a DDE and seeking its solution, we are given a discrete set of solution values and try to determine the corresponding DDE. In practice, however, parameter estimation involves solving a parameter-dependent DDE many times whilst changing the parameter values in some “intelligent” manner. Implicit in the problem is the assumption that we have some idea of the form of the DDE – the more accurate our model, the more likely we are to obtain a “good fit” to the data. Mathematically, the parameter estimation problem may be summarized as:

Given data $\{\sigma_i, Y(\sigma_i)\}$ and the corresponding solution values of the parameter-dependent DDE $\{y(\sigma_i; \mathbf{p})\}$, we seek to minimize over the parameters \mathbf{p} the objective function

$$\Phi(\mathbf{p}) := \sum_i (Y(\sigma_i) - y(\sigma_i; \mathbf{p}))^2.$$

For DDEs, the parameters may occur in the initial function $\Psi(t; \mathbf{p})$ (and the initial values if $y(t_0) \neq \Psi(t_0; \mathbf{p})$), the delay functions $\{\alpha_i(t; \mathbf{p})\}$, the derivative function $f(t; \mathbf{p})$ and possibly even in the positioning of the initial point $t_0(\mathbf{p})$. (In some cases, due to the propagation of discontinuities in DDEs, this can lead to problems with the smoothness of $\Phi(\mathbf{p})$ [3].)

Archi can handle all of these possibilities, and this is achieved by extending the list of user-callable routines². (A minor change is also required to be made to each of the user-supplied routines, allowing the parameters to be available to the user.) There are two choices of optimization routine for use with the extended *Archi*: LMDIF for use with *Archi*-L (which includes a modified version of LMDIF), and E04UPF for use with *Archi*-N. LMDIF is an unconstrained optimization routine based on the Levenberg-Marquardt algorithm and is available from NETLIB. E04UPF is a non-linear constrained optimization routine and is available in the NAG library. Both routines require an approximation to the Jacobian matrix of the objective function $\Phi(\mathbf{p})$, and this is calculated using forward-differences.

7.1 Specifying the Number of Parameters

The *number of parameters* to be estimated is specified by:

```
CALL PDIMEN(INTGRS, REALS, number)
```

where `number` > 0 . Once set, the number cannot be changed and it is *most important* that `number` does not exceed the actual number of parameters used (otherwise the minimization routine may fail).

²However, additional parameters specific to the NAG routine may also be set in the subroutine MINIM, but for most problems this should be unnecessary.

7.2 Specifying the Parameter Tolerance

The *parameter tolerance*, the maximum size of $\|Y(t) - y(t; \mathbf{p})\|_2$ before the parameters \mathbf{p} are accepted, is specified by:

```
CALL PARTOL(INTGRS,REALS,tolerance)
```

where *tolerance* must not be ‘too small’ ($tolerance \geq 100\mu$). Ideally, *tolerance* should not be less than *eps*, the error-per-step tolerance used for solving the DDE (§1.3), so as to avoid “fitting the noise” in the solution. Once set, *tolerance* cannot be changed.

7.3 Specifying the Initial Point as a Parameter

After the number of parameters has been set (§7.1), the position of the initial point t_0 may be specified as being a parameter by:

```
CALL PARTO(INTGRS,REALS,parameter)
```

where *parameter* corresponds to a valid parameter. Once set, the parameter mapping cannot be changed or deleted. For those components of the solution that are continuous at the ‘original’ initial point t_{START} (§1.1, §1.11.2), the initial values $y_i(t_0(\mathbf{p}))$ are automatically specified as having the values $\Psi_i(t_0(\mathbf{p}); \mathbf{p})$; however see §7.4. If discontinuity tracking (§1.11.1) has been selected and discontinuities have been specified as occurring at t_{START} , then the continuity record is automatically updated as the position of $t_0(\mathbf{p})$ changes.

7.4 Specifying an Initial Value as a Parameter

After the number of parameters has been set (§7.1), but *before* the sample data has been read (§7.5), it is possible to specify an initial value as being a parameter. This is achieved by:

```
CALL PARMAP(INTGRS,REALS,parameter,component)
```

where *parameter* must be a valid parameter and *component* must be a valid solution component. After an initial value has been specified as being a parameter, the mapping to *component* cannot be changed or deleted. If an initial value is specified to be a parameter, then the parameter value takes precedence over the initial value specified in the driver program and over the solution value implied by the initial function and the continuity record at the initial point (see §7.3).

7.5 Specifying the Data to be Fitted

The data to be fitted is specified in a text file, the precise format of which is given in the two examples below. The data is read by:

```
CALL DATUM(INTGRS,REALS,filename)
```

The first line of *filename* must be a space-separated, integer list that contains the number of data to be fitted for each solution component. Next the sample data is specified, with one data pair $(\sigma_i, Y(\sigma_i))$ per line. Sets of data corresponding to different solution components must be separated by a single blank line.

7.6 Solving the Minimization Problem

After the parameter estimation problem has been specified, it is solved by:

```
CALL MINIM(INTGRS,REALS,PARAM,Y)
```

where *PARAM* is the array of (initial) parameter values and *Y* is the array of (initial) solution values. The “best-fit” parameter values \mathbf{p}^* are returned in *PARAM* if the minimization is successful.

7.7 Printing the Results

In addition to being able to print out the “best-fit” solution using *Archi*’s normal output routines (§1.10.2, §1.10.4), the components of the residual vector and the 2-norm of the residual vector can also be printed. (The number of iterations of the minimization routine is available in *INTGRS(72)*.)

7.7.1 Printing the Residual Vector

The data points $\{\sigma_i\}$ and the corresponding components of the residual vector $\{Y(\sigma_i) - y(\sigma_i; \mathbf{p})\}$ can be printed by:

```
CALL PRNTRX(INTGRS,REALS)
```

after the parameter estimation problem has been solved.

7.7.2 Printing the 2-Norm of the Residual Vector

The 2-norm of the residual vector for the “best-fit” parameter values can be determined by calling the double precision function:

```
RESID(INTGRS,REALS)
```

after the parameter estimation problem has been solved.

7.8 Constrained Optimization Problems

The occurrence of a constraint in a DDE parameter estimation problem arises naturally from trying to estimate a lag function, since $\tau(t; \mathbf{p})$ must always be non-negative. However, constrained optimization problems are, computationally, much more expensive to solve than unconstrained optimization problems.

The simplest case is that of estimating a constant lag $\tau(t; \mathbf{p}) \equiv \mathbf{p}_i$. It may be regarded as being either a constrained optimization problem with a simple bound on the parameter ($0 \leq \mathbf{p}_i < \infty$), or an unconstrained optimization problem with $\tau(t; \mathbf{p}) \equiv \mathbf{p}_i^2$. However, for a time-varying (or state-dependent) lag, the situation can be much more complicated: Optimization routines usually allow for linear (and non-linear) constraints on the parameter values, such as $-1 \leq 2\mathbf{p}_1 + 3\mathbf{p}_2 \leq 3$ (and $0 \leq \mathbf{p}_1^2 - \cos(\mathbf{p}_2) \leq 2$) say, but when estimating a lag function $\tau(t)$, the required constraint is $\tau(t; \mathbf{p}) \geq 0$ for all t in the range of integration. This type of non-linear *functional* constraint problem cannot be solved using E04UPF. One approach to estimating an unknown lag function is to approximate it using linear splines [4, 9]. Thus the non-linear *functional* constraint problem reduces to ensuring that the lag is non-negative at the support points – a simple “bounds on the parameters” problem. However, due to the resulting low-order discontinuities in the approximation to the lag function, this approach can have its drawbacks [3].

Archi-N allows simple bounds on the parameters to be specified (§7.8.3), and more complex linear and non-linear constraints (§7.8.1, §7.8.2). (In order to allow the same driver program to be used with both *Archi-L* and *Archi-N*, *Archi-L* includes equivalent dummy constraint routines.)

7.8.1 Specifying the Number of Linear/Non-Linear Constraints

The number of linear/non-linear constraints is specified by:

```
CALL LIMITS(INTGRS,REALS,constraints)
```

where `constraints` ≥ 0 . This must be done before any simple parameter bounds are specified (§7.8.3) *even* if `constraints` = 0. Once set, the number of constraints cannot be changed.

7.8.2 Specifying a Linear/Non-Linear Constraint Bound

After the number of linear/non-linear constraints has been specified (§7.8.1), it is possible to set the lower and upper bound on a constraint by:

```
CALL NBOUND(INTGRS,REALS,constraint,lower,upper)
```

where `constraint` must be a valid linear/non-linear constraint and *lower* < *upper*.

7.8.3 Specifying a Bound on a Parameter

A lower and upper bound may be specified for a parameter, after the number of parameters (§7.1) and the number of constraints (§7.8.1) has been set, by:

CALL BOUND(INTGRS,REALS,parameter,lower,upper)

where parameter must be a valid parameter and $lower < upper$. Once set, the bounds may be changed but they cannot be deleted.

8 Writing a Linear/Non-Linear Constraint Routine

The linear/non-linear constraints for use with E04UPF are specified in a subroutine. The **mandatory skeleton** is:

```
SUBROUTINE CONSTR(INTGRS,REALS,PARAM,CFNS)
  INTEGER INTGRS(*)
  DOUBLE PRECISION CFNS(*),PARAM(*),REALS(*)
  CFNS(1) = .....
  RETURN
END
```

where PARAM is the array of parameter values and CFNS is the array of constraint values for the current parameter values.

Sample Parameter Estimation Driver Programs and Results

This section contains two sample driver programs for *Archi-L* and *Archi-N*. In both examples, the sample data was obtained using *Archi* with an error-per-step tolerance of 10^{-12} . Although these examples are included to help check whether *Archi-L* and *Archi-N* are working correctly, differences in the numerical results should be expected.

Example 1

$$\begin{aligned} y'(t) &= p_1 y(t - p_2) & (t \geq 0), \\ \Psi(t) &= t^2 & (t \leq 0), \end{aligned}$$

where $p = [-1, 2]$.

Sample data in file 'data1' produced by `sample1.f`:

```
5
2.0000000000000000    -2.6666666666666661
4.0000000000000000    1.3333333333333317
6.0000000000000000    3.4666666666666748
8.0000000000000000    -2.7555555555558588
10.0000000000000000   -4.2285714285729803

PROGRAM PAREST
DOUBLE PRECISION REALS(50000),PARAM(3),RESID,Y(1)
INTEGER INTGRS(2000)
CALL INIT(INTGRS,REALS,2000,50000)
CALL RANGE(INTGRS,REALS,0D0,10D0)
CALL ERROR(INTGRS,REALS,1,1D-9)
CALL REPORT(INTGRS,REALS,0)
CALL STATS(INTGRS,REALS,0)
CALL PDIMEN(INTGRS,REALS,3)
CALL PARTOL(INTGRS,REALS,1D-3)
CALL PARMAP(INTGRS,REALS,3,1)
CALL LIMITS(INTGRS,REALS,1)
CALL NBOUND(INTGRS,REALS,1,-3D0,3D0)
CALL BOUND(INTGRS,REALS,2,0D0,10D0)
CALL DATUM(INTGRS,REALS,'data1')
CALL HSTART(INTGRS,REALS,1D0)
CALL TRACK(INTGRS,REALS,2)
CALL SOFT(INTGRS,REALS,1,1,1)
CALL CONTY(INTGRS,REALS,1,1,0D0)
Y(1) = 1D0
PARAM(1) = -2D0
PARAM(2) = 3D0

3 parameters to fit
Parameter tolerance 0.001
y(0) is the third parameter
1 non-linear constraint (E04UPF only)
-3 <= constraint <= 3 (E04UPF only)
Bound the delay (E04UPF only)
Read in sample data

PARMAP overwrites this value
Set initial parameter values
```

```

PARAM(3) = 1D0
CALL MINIM(INTGRS,REALS,PARAM,Y)
WRITE (*,*) PARAM
WRITE (*,*) RESID(INTGRS,REALS)
END

SUBROUTINE DERIV(INTGRS,REALS,PARAM,X,F)
DOUBLE PRECISION DELAY,F(*),NEUTRL,PARAM(*),REALS(*),VOLTRA,X
INTEGER INTGRS(*)
F(1) = PARAM(1)*DELAY(INTGRS,REALS,1,1)
RETURN
END

SUBROUTINE CONSTR(INTGRS,REALS,PARAM,CFNS)
INTEGER INTGRS(*)
DOUBLE PRECISION CFNS(*),PARAM(*),REALS(*)
CFNS(1) = PARAM(1)*PARAM(2)
RETURN
END

DOUBLE PRECISION FUNCTION LAG(INTGRS,REALS,PARAM,LAGFN,X)
DOUBLE PRECISION DELAY,NEUTRL,PARAM(*),REALS(*),X
INTEGER INTGRS(*),LAGFN
LAG = X-PARAM(2)
RETURN
END

DOUBLE PRECISION FUNCTION YINIT(PARAM,SOLN,X)
DOUBLE PRECISION PARAM(*),X
INTEGER SOLN
YINIT = X*X
RETURN
END

```

Solve minimization problem
Print best-fit parameter values
Print 2-norm of residual

Specify constraint (E04UPF only)
Constraint is $-3 \leq p_1 p_2 \leq 3$
(See above for bounds)

Delay is second parameter

The results produced by `fit1.f` are:

Results for LMDIF

```

-1.0000000356398   1.9999999777041   -3.2273854214495D-07
 1.3778193366560D-06

```

Results for E04UPF – constraint *inactive* at p^*

```

-0.99973817604622   2.0000017908938   -1.8713608408679D-03
 1.7194976371510D-03

```

Example 2

$$\begin{aligned}
 y_1'(t) &= p_1 y_2(t-1) & (t \geq p_4), \\
 y_2'(t) &= p_2 y_1(t-2) & (t \geq p_4), \\
 \Psi_1(t) &= p_3 t^2 & (t \leq p_4), \\
 \Psi_2(t) &= 1-t & (t \leq p_4),
 \end{aligned}$$

where $p = [-1, 2, 1, 0]$.

Sample data in file 'data2' produced by `sample2.f`:

```

3 5
2.0000000000000000   -5.333333333333410
6.0000000000000000   -10.805555555558749
10.0000000000000000  310.16329365079110

2.0000000000000000   6.333333333333339
4.0000000000000000   -1.400000000000070
6.0000000000000000   -47.16666666666650
8.0000000000000000   -118.07142857142843
10.0000000000000000  -11.769929453255918

PROGRAM PAREST
DOUBLE PRECISION REALS(50000),PARAM(4),Y(2)
INTEGER INTGRS(2000)

```

```

CALL INIT(INTGRS,REALS,2000,50000)
CALL RANGE(INTGRS,REALS,0D0,10D0)
CALL DIMEN(INTGRS,REALS,2)
CALL DELAYS(INTGRS,REALS,2)
CALL ERROR(INTGRS,REALS,1,1D-9)
CALL REPORT(INTGRS,REALS,0)
CALL STATS(INTGRS,REALS,0)
CALL PDIMEN(INTGRS,REALS,4)
CALL PARTOL(INTGRS,REALS,1D-9)
CALL PARTO(INTGRS,REALS,4)
CALL LIMITS(INTGRS,REALS,0)
CALL BOUND(INTGRS,REALS,2,0.5D0,1.5D0)
CALL DATUM(INTGRS,REALS,'data2')
CALL HSTART(INTGRS,REALS,1D0)
CALL TRACK(INTGRS,REALS,2)
CALL SOFT(INTGRS,REALS,1,2,2)
CALL SOFT(INTGRS,REALS,2,1,1)
CALL CONTY(INTGRS,REALS,1,1,0D0)
CALL CONTY(INTGRS,REALS,2,1,0D0)
Y(1) = 0D0
Y(2) = 1D0
PARAM(1) = -0.95D0
PARAM(2) = 1.95D0
PARAM(3) = 0.95D0
PARAM(4) = 0.1D0
CALL MINIM(INTGRS,REALS,PARAM,Y)
WRITE (*,*) PARAM
WRITE (*,*) 'Iterations = ',INTGRS(72)
CALL PRNTRX(INTGRS,REALS)
END

SUBROUTINE DERIV(INTGRS,REALS,PARAM,X,F)
DOUBLE PRECISION DELAY,F(*),NEUTRL,PARAM(*),REALS(*),VOLTRA,X
INTEGER INTGRS(*)
F(1) = PARAM(1)*DELAY(INTGRS,REALS,1,2)
F(2) = PARAM(2)*DELAY(INTGRS,REALS,2,1)
RETURN
END

SUBROUTINE CONSTR(INTGRS,REALS,PARAM,CFNS)
INTEGER INTGRS(*)
DOUBLE PRECISION CFNS(*),PARAM(*),REALS(*)
RETURN
END

DOUBLE PRECISION FUNCTION LAG(INTGRS,REALS,PARAM,LAGFN,X)
DOUBLE PRECISION DELAY,NEUTRL,PARAM(*),REALS(*),X
INTEGER INTGRS(*),LAGFN
GOTO (1,2) LAGFN
1 LAG = X-1D0
RETURN
2 LAG = X-2D0
RETURN
END

DOUBLE PRECISION FUNCTION YINIT(PARAM,SOLN,X)
DOUBLE PRECISION PARAM(*),X
INTEGER SOLN
GOTO (1,2) SOLN
1 YINIT = PARAM(3)*X*X
RETURN
2 YINIT = 1D0-X
RETURN
END

```

4 parameters to fit
Parameter tolerance 0.000000001
t0 is fourth parameter
No linear/non-linear constraints
Bounds on second parameter (E04UPF only)
Read in sample data

Specify continuity at t0(p)
Specify continuity at t0(p)
Initial function overwrites this value
Initial function overwrites this value
Set initial parameter values

Solve minimization problem
Print best-fit parameter values
Print number of minimization iterations
Print the residual vector

No constraints specified

The results produced by fit2.f are:

Results for LMDIF

```

-1.0000000010278    2.0000000433762    1.0000000187338    3.2497202068459D-08
Iterations = 9
Residual vector =
2.0000000000000000    -0.23745930999297116E-06
6.0000000000000000    0.86085815809155974E-07
10.0000000000000000   -0.62002698086871533E-07

```

2.0000000000000000	0.76891684130941940E-07
4.0000000000000000	-0.21635309410683590E-06
6.0000000000000000	-0.44379196850741209E-06
8.0000000000000000	0.31690576918208535E-06
10.0000000000000000	-0.96793099757519485E-07

Results for E04UPF – parameter bound *active* at p^*

```

** Current point cannot be improved upon.
** ABNORMAL EXIT from NAG Library routine E04UPF: IFAIL =      6
** NAG soft failure - control returned
  -1.0503319877505    1.5000000000000    1.5172010479580   -0.16339720578440
Iterations =      3
Residual vector =
  2.0000000000000000    3.0166073105172782
  6.0000000000000000    10.593462725836309
 10.0000000000000000   -57.881152566850062
  2.0000000000000000   -2.4734015738519144
  4.0000000000000000   -1.2943779400742674
  6.0000000000000000    4.9758394050333976
  8.0000000000000000    18.661680544620040
 10.0000000000000000    59.573672493972190

```

References

- [1] C.T.H. BAKER & C.A.H. PAUL, *Parallel continuous Runge-Kutta methods and vanishing-lag delay differential equations*, Adv. Comp. Math., 1 (1993), pp. 367–394.
- [2] C.T.H. BAKER & C.A.H. PAUL, *A global convergence theorem for a class of parallel continuous explicit Runge-Kutta methods and vanishing-lag delay differential equations*, SIAM J. Numer. Anal., 33 (1996), pp. 1559–1576.
- [3] C.T.H. BAKER & C.A.H. PAUL, *Pitfalls in parameter estimation for delay differential equations*, SIAM J. Sci. Comp., 18 (1997), pp. 305–314.
- [4] H.T. BANKS, J.A. BURNS & E.M. CLIFF, *Parameter Estimation and Identification for Systems with Delays*, SIAM J. Control Optim., 19 (1981), pp. 791–828.
- [5] J.R. DORMAND & P.J. PRINCE, *A family of embedded Runge-Kutta formulae*, J. Comp. Appl. Math., 6 (1980), pp. 19–26.
- [6] M.A. FELDSTEIN & K.W. NEVES, *High-order methods for state-dependent delay differential equations with non-smooth solutions*, SIAM J. Numer. Anal., 21 (1984), pp. 844–863.
- [7] S. FILIPPI & U. BUCHACKER, *Stepsize control for delay differential equations using a pair of formulae*, J. Comp. Appl. Math., 26 (1989), pp. 339–343.
- [8] I. GLADWELL, L.F. SHAMPINE, L.S. BACA & R.W. BRANKIN, *Practical aspects of interpolation in Runge-Kutta codes*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 322–341.
- [9] K.A. MURPHY, *Estimation of Time- and State-Dependent Delays and Other Parameters in Functional Differential Equations*, SIAM J. Appl. Math., 50 (1990), pp. 972–1000.
- [10] C.A.H. PAUL, *A fast, efficient, low-storage adaptive quadrature scheme*, MCCM report 213 (1992), ISSN 1360-1725.
- [11] L.F. SHAMPINE, *Interpolation for Runge-Kutta methods*, SIAM J. Numer. Anal., 22 (1985), pp. 1014–1026.
- [12] L.F. SHAMPINE, *Some practical Runge-Kutta formulas*, Math. Comp., 46 (1986), pp. 135–150.
- [13] D.R. WILLÉ & C.T.H. BAKER, *A short note on the propagation of derivative discontinuities in Volterra-delay integro-differential equations*, MCCM report 187 (1990), ISSN 1360-1725.
- [14] D.R. WILLÉ & C.T.H. BAKER, *The tracking of derivative discontinuities in systems of delay differential equations*, Appl. Num. Math., 9 (1992), pp. 209–222.