European Society of Computational Methods in Sciences and Engineering (ESCMSE)

Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)
vol. 3, no. 1-2, 2008, pp. 61-80
ISSN 1790–8140

Solving Differential Algebraic Equations by Taylor Series (III): the DAETS Code[1]

Nedialko S. Nedialkov [2]

Department of Computing and Software, McMaster University,
Hamilton, Ontario, L8S 4L7, Canada

John D. Pryce [3]

Department of Information Systems, Cranfield University,
RMCS Shrivenham, Swindon SN6 8LA, UK

Received 30 October, 2007; accepted in revised form 26 January, 2008

Abstract: The authors have developed a Taylor series method for solving numerically an initial-value problem differential algebraic equation (DAE) that can be of high index, high order, nonlinear, and fully implicit, see BIT 45:561–592, 2005 and BIT 41:364-394, 2001. Numerical results have shown this method to be efficient and very accurate, and particularly suitable for problems that are of too high an index for present DAE solvers. This paper outlines this theory and describes the design, implementation, usage and performance of DAETS, a DAE solver based on this theory and written in C++.

© 2008 European Society of Computational Methods in Sciences and Engineering

Keywords: Differential algebraic equations (DAEs), structural analysis, Taylor series, automatic differentiation

Mathematics Subject Classification: 34A09, 65L80, 65L05, 41A58

1 Introduction

1.1 What DAETS does and the tools it uses

This paper describes the structure and use of a code DAETS (Differential Algebraic Equations by Taylor Series) that solves initial value problems for differential algebraic equation systems (DAEs) for state variables $x_j(t),\ j = 1, \ldots, n$, of the general form

$$f_i(\,t, \text{ the } x_j \text{ and derivatives of them}\,) = 0, \quad i = 1, \ldots, n, \tag{1}$$

by expanding the solution in a Taylor series (TS) at each integration step. The f_i can be arbitrary expressions built from the x_j and t using $+, -, \times, \div$, other analytic standard functions, and the

[1]Published electronically March 31, 2008
[2]Corresponding author. E-mail: nedialk@mcmaster.ca
[3]E-mail: j.d.pryce@ntlworld.com

differentiation operator d^p/dt^p. They can be nonlinear and fully implicit in the variables and derivatives. An equation such as

$$\frac{((tx_1')')^2}{1+(x_2')^2} + t^2 \cos x_2 = 0 \tag{2}$$

can be encoded directly into DAETS. Derivatives need not actually be present, so DAETS can solve continuation problems $\mathbf{f}(t, \mathbf{x}) = \mathbf{0}$, taking t as the continuation parameter. It has handled difficult problems of this kind, as reported in Subsection 5.3.

A common measure of the numerical difficulty of a DAE is its *differentiation index* ν_d, the number of times the f_i must be differentiated (w.r.t. t) to obtain equations that can be solved to form an ODE system for the x_j. Index ≥ 3 is normally considered hard. DAETS is not inherently affected by high index, as explained later. We report results on up to index-47 DAEs.

One of the hardest parts of DAE solution can be finding an initial consistent point. This can be seen as a minimization problem, and DAETS gives this task to a proven optimization package IPOPT [28], which has proved robust and effective.

Stiff behaviour can be present for DAEs, as for ODEs. DAETS, like other codes based on Taylor series, handles moderate stiffness well but is unsuitable for highly stiff problems.

DAETS is not a validating code that finds guaranteed enclosures of the exact solution. The authors know in principle how to implement such a code, but it is a large project.

DAETS is written in C++. Apart from IPOPT, it uses Stauning's automatic differentiation (AD) package FADBAD++ [27] and the C version of Volgenant's LAP [11] code for Linear Assignment Problems.

The rest of this introduction takes a software developer's viewpoint. Subsection 1.2 reviews theoretical approaches to DAEs based on derivatives of equations, as is that of DAETS; and describes the origins of the software architecture of DAETS. Subsection 1.3 gives a broad overview of the numerical method, and discusses the impact it has on the architecture and the user interface.

Following this, Section 2 outlines the structural analysis (SA) theory on which the numerical method is based. Section 3 describes the main components of the algorithm. Section 4 discusses the user-visible classes, and other user interface matters. Section 5 reports and discusses the performance of the code on various problems. Section 6 shows a simple complete program using DAETS. Section 7 summarises our experience with DAETS and describes some outstanding problems.

1.2 Background

Many problems of importance in science and industry are modelled as DAEs. Examples from applications are in the Test Set for Initial Value Solvers [14], which contains problems from electronic circuits, mechanical systems and chemical processing. One reason why higher index DAEs are of interest because adding detail to a model, such as accounting for internal behaviour of actuating motors in a robot arm, often raises the index.

For a traditional solver, the problem is typically written as a first-order ODE $\mathbf{x}' = \mathbf{f}(t, \mathbf{x})$ or first-order DAE $M(t, \mathbf{x})\mathbf{x}' = \mathbf{f}(t, \mathbf{x})$, where M is a singular matrix. The code only sees a "black box" routine that computes the value of $\mathbf{f}$ or M or (for stiff problems) Jacobian information at given inputs $(t, \mathbf{x})$. It has long been known in various theoretical frameworks that a DAE can be seen as an ODE on a manifold and can be accurately solved as such, provided one "knows" how to manipulate derivatives of the defining functions. This underlies the *jet space* approach [25] based on differential geometry, Campbell's *derivative array* theory [4] and more practical approaches of *index reduction* [1, 13]. DAETS is based on Pryce's *structural analysis* [24], which has this same basic approach and is itself an extension of the method of Pantelides [19].

As regards using Taylor series to solve DAEs, we believe this was first done by Chang [5], in an *ad hoc* way. The software architecture of DAETS is derived from that of Nedialkov's VNODE code

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

[15] mainly written during his PhD project at the University of Toronto. VNODE finds *validated* solutions (that is, enclosures of the solutions) of ordinary differential equation (ODE) initial value problems, using interval arithmetic. More important than the interval aspect is that VNODE is an object-oriented implementation of a general design for ODE solvers due to Hull and Enright [10]. This design separates parts of the algorithm into modules, such as the stepping formula, error estimation, step size choice, output to the user, etc., in a way that makes it easy to change the algorithm for one part with minimal impact on the others.

DAETS and VNODE have in common parts of the implementation as well as of the design: for instance they both use FADBAD++ to do AD, and VNODE has an optional stepping module that uses TS. However, DAETS handles the implicit system (1) while VNODE handles an explicit ODE system $\mathbf{x}' = \mathbf{f}(t, \mathbf{x})$, which makes a large difference to the details of how the two codes use FADBAD++ and how they generate Taylor coefficients (TCs). Some significant parts of the DAETS algorithm have no counterpart in the Hull–Enright design.

1.3 The method and how it affects the design

We describe in broad terms the numerical method based on Pryce's *structural analysis* (SA) [23, 24], and discuss how its features affect the design of the code, especially the user interface. In this subsection, "software" means our (or another) DAE solver, as opposed to the infrastructure tools below it or an application on top of it.

The numerical method. Starting with code for the functions f_i in (1), an SA-based method uses automatic differentiation (AD) to evaluate suitable derivatives $\mathrm{d}^r/\mathrm{d}t^r$ of the f_i. By equating these to zero at a given $t = \widehat{t}$, it solves implicitly for derivatives, or equivalently TCs, of the solution components $x_j(t)$ at $\widehat{t}$.

The AD can be handled in many ways, depending on language features and on the tools one considers efficient and convenient. We use Ole Stauning's popular AD tool, FADBAD++, which uses C++ templates and operator overloading to compute derivatives. There is evidence that other tools may give somewhat faster code, e.g. ADOL-C [7], but we have found FADBAD++ is convenient for the software writer, allows a straightforward user interface, and has to date caused no problems with installing the software.

As illustrated in (2), differentiation $\mathrm{d}/\mathrm{d}t$ can be used anywhere in the expressions defining the DAE. To enable this, Stauning modified FADBAD++ to make $\mathrm{d}/\mathrm{d}t$ a "first-class" operator (named `diff`), like the arithmetic operations and standard functions. Also, to make over/underflow less likely in computing coefficients a_r of a Taylor expansion $a(t^* + h) = \sum_r a_r h^r$, DAETS computes the terms $a_r h^r$ directly. This is neatly done by an implicit change of independent variable from t to s, where $t = t^* + sh$. This required creating our version of the $\mathrm{d}/\mathrm{d}t$ operator (named `Diff`) inside DAETS, so FADBAD++ is not changed.

Various numerical methods can be based on SA. "Solving implicitly" can in effect reduce the DAE to an ODE system — numerically implementing the definition of the differentiation index — and a standard method such as Runge–Kutta or BDF can be applied to the result. Methods of this kind are starting to be studied. We have chosen the approach in Pryce's original papers, using the AD to evaluate the TS to some order. The resulting method is generally not as efficient as standard DAE solvers on problems they can solve, but becomes increasingly efficient at high accuracies and is relatively simple to code.

In the SA one computes the $n \times n$ *signature matrix*, and $2n$ integers, the *offsets* of the variables and of the equations. These prescribe the overall process for computing TCs, as well as how to form the *System Jacobian* $\mathbf{J}$ (5) that is central to the theory and the numerical method.

The TCs are used with an appropriate stepsize to find a truncated TS approximation of the solution, which is then *projected* to satisfy the constraints of the DAE. The process is repeated on

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

each integration step in a standard time-stepping manner. Error estimation and step size selection are like that in Taylor codes for ODEs, but the offsets complicate the details.

If $\mathbf{J}$ is nonsingular at a *consistent point*, see Subsection 2.3, the SA has succeeded, and the DAE is solvable in a neighborhood of this point. Although SA applies to a wide range of DAEs, there are problems on which it fails: when $\mathbf{J}$ is singular at a point at which the DAE is solvable. Typically this happens when the equations (1) are "not sparse enough" to reveal the underlying structure of the system. Examples are discussed in [16, 23, 24].

An SA-based method derives a *structural index* ν_s, which is the same as the index found by the method of Pantelides [19] for DAEs to which that method applies. It is shown by Reißig, Martinson and Barton [26] that ν_s can be arbitrarily greater than ν_d; but in [24, Subsection 5.3], that provided the SA succeeds — that is, $\mathbf{J}$ is nonsingular — ν_s can never be less than ν_d, and in [16] that overestimating ν_s causes, at worst, mild inefficiency in the solution process. Moreover the method is robust in the sense that it can always detect (up to roundoff) when $\mathbf{J}$ is singular, and therefore indicate that the SA fails: see [16, Algorithm 6.1].

Effect on the user interface. C++ code for the functions f_i in (1) looks much as for a standard ODE solver for $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$. However, the active variables must be declared of a *templated* type, instantiated at compile time with several different actual types. One type is used for the numerical AD, another to compute the signature matrix, and so on. This imposes some restrictions on the allowed expressions. Also, the differentiation operator `Diff(·,q)` denoting $\mathrm{d}^{\mathtt{q}}/\mathrm{d}t^{\mathtt{q}}$ can be applied to any *active* item. The latter are the inputs `x[j]` denoting x_j , the outputs `f[i]` denoting f_i, and any code variables or expressions on the computational path between these.

When the SA method succeeds, it gives the user much useful information about the structure of the DAE. We provide a way for the calling program to print the signature matrix, offsets and related data after the SA has been done. In fact the user *needs* to know SA data to use an SA-based method effectively, because the offsets determine the shape of the set of initial values to be given to the solver (*fixed* values, or *free*, guessed, values). For traditional ODE/DAE solvers, the initial values are flat vectors: here they form an irregular array, see page 71.

Coding the calling program for an unfamiliar DAE tends to be a two-stage process. First, give the function code to DAETS and make it print a description of the structure. Setting initial values can then be coded correctly, although it may need thought about their physical meaning to get a sensible choice of fixed and free values.

The fact that analysis precedes numerical solution affects the user interface. First, it is why the interface makes two (main) classes available. One, `DAEsolution`, holds the irregular array of x_j-and-derivatives at a given t, plus related data. The other, `DAEsolver`, knows among other things the result of the structural analysis. Its `integrate` method can be applied to different `DAEsolution` objects. Thus one can compute many solution paths without re-doing the SA.

Second, each stage has its own kind of error. The SA can find there is no finite transversal (Subsection 2.1), indicating an ill-posed problem. Or the SA can appear to succeed, but the resulting Jacobian matrix $\mathbf{J}$ is always singular, showing the SA doesn't reveal the true structure of the DAE. Or the code may find a nonsingular $\mathbf{J}$ but be unable to find a consistent point. Or it may find a consistent point and start along the solution path, but grind to a halt for some reason.

The two-way information flow in preparing the problem for solution gives DAETS some features of a Problem Solving Environment. We do not offer an interactive GUI yet, but this would be useful. There is a contrasting requirement, however. The software may be needed as an "engine", to solve a DAE of known structure as part of a larger application. For such use, it must suppress printing and return any error diagnostics by a flag (or similar) that is handled by the calling program. DAETS can be used in such a silent mode as well as in a verbose one.

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

2 Theory

2.1 Pryce's structural analysis of DAEs

We present as much of Pryce's structural analysis [24] as is needed for this paper.

A *transversal* T of an $n \times n$ matrix (σ_{ij}) is a set of n positions in the matrix with one entry in each row and each column — a set $\{(1, j_1), (2, j_2), \ldots, (n, j_n)\}$ where $(j_1, \ldots, j_n)$ are a permutation of $(1, \ldots, n)$. The *value* of T is $\operatorname{Val} T = \sum_{(i,j)\in T} \sigma_{ij}$.

Given a DAE in the form of (1), we perform the following steps.

1. Form the $n \times n$ *signature matrix* $\Sigma = (\sigma_{ij})$, where

$$\sigma_{ij} = \begin{cases} \text{order of the derivative to which the } j\text{th variable } x_j \text{ occurs in} \\ \text{the } i\text{th equation } f_i\text{; or} \\ -\infty \text{ if } x_j \text{ does not occur in } f_i. \end{cases}$$

2. Find a *highest value transversal* (HVT), which is a transversal T that makes $\operatorname{Val} T$ as large as possible. This value is also, by definition, the *value of the signature matrix*, written $\operatorname{Val} \Sigma$.

   The value of any transversal, and of Σ, is either an integer or $-\infty$. The DAE is *structurally regular* if $\operatorname{Val} \Sigma$ is finite: that is, if at least one transversal exists, all of whose σ_{ij} are finite. Otherwise it is *structurally ill-posed* — probably the problem is wrongly posed in some way.

3. Find n-dimensional integer vectors $\mathbf{c}$ and $\mathbf{d}$, with all $c_i \geq 0$, that satisfy

$$d_j - c_i \geq \sigma_{ij} \quad \text{for all } i, j = 1, \ldots, n \text{ and} \tag{3}$$
$$d_j - c_i = \sigma_{ij} \quad \text{for all } (i, j) \in T. \tag{4}$$

   By [24, Lemma 3.3], if a transversal T and vectors $\mathbf{c}, \mathbf{d}$ are found such that (3) and (4) hold, then necessarily T is a HVT. Summing (4) over T gives an alternative formula for $\operatorname{Val} \Sigma$:

$$\sum_{j=1}^{n} d_j - \sum_{i=1}^{n} c_i = \operatorname{Val} T = \operatorname{Val} \Sigma.$$

   It follows that for any $\mathbf{c}$ and $\mathbf{d}$, if (3, 4) hold for some HVT, then (4) holds for any HVT.

   The vectors $\mathbf{c}$ and $\mathbf{d}$ are the *offsets*. They are never unique. It is a little more efficient, but not necessary, to choose the *canonical* offsets, which are the smallest in the sense of $\mathbf{a} \leq \mathbf{b}$ if $a_i \leq b_i$ for each i.

4. Form the $n \times n$ System Jacobian matrix

$$\mathbf{J} = \frac{\partial \left(f_1^{(c_1)}, \ldots, f_n^{(c_n)} \right)}{\partial \left(x_1^{(d_1)}, \ldots, x_n^{(d_n)} \right)}. \tag{5}$$

   By results in [24], equation (5) has the equivalent formulation:

$$\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j^{(d_j - c_i)}} = \begin{cases} \dfrac{\partial f_i}{\partial x_j^{(\sigma_{ij})}} & \text{if } d_j - c_i = \sigma_{ij}, \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

5. Seek values for the x_j and for appropriate derivatives, consistent with the DAE in the sense of Subsection 2.3, at which $\mathbf{J}$ is nonsingular. If such values exist, they define a point through which there is locally a unique solution of the DAE. In this case we say the method "succeeds".

Step (2) is a Linear Assignment Problem (LAP), a form of a linear programming problem for which good software exists [11]. Step (3) defines its dual. The two formulae for $\operatorname{Val}\Sigma$ instance the fact that primal and dual have the same optimal value.

When the method succeeds:

- $\operatorname{Val}\Sigma$ equals the number of *degrees of freedom* (DOF) of the DAE, that is the number of independent initial conditions required.
- An upper bound for the differentiation index ν_d is given by the *Taylor index*

$$\nu_T = \max_i c_i + \begin{cases} 1 & \text{if some } d_j \text{ is zero,} \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

  In many cases, $\nu_T = \nu_d$.

2.2 An example

In this paper, we give examples based on the simple pendulum, a DAE of differentiation-index 3. Although it is small, solving it with our method displays almost all the algorithmic features. It is:

$$\begin{aligned} 0 &= f = x'' + x\lambda \\ 0 &= g = y'' + y\lambda - G \\ 0 &= h = x^2 + y^2 - L^2. \end{aligned} \tag{8}$$

Here gravity G and length L of pendulum are constants, and the dependent variables are the coordinates $x(t)$, $y(t)$ and the Lagrange multiplier $\lambda(t)$.

A signature matrix Σ may be shown by a "tableau", which annotates it with the offsets c_i, d_j and the function and variable names, and marks the positions of a HVT. For (8), there are two HVTs, marked $^\bullet$ and $^\circ$ in the tableau below. The canonical offsets are $\mathbf{c} = (0, 0, 2)$ and $\mathbf{d} = (2, 2, 0)$:

$$\begin{array}{c|ccc|c} & x & y & \lambda & c_i \\ f & 2^\bullet & -\infty & 0^\circ & 0 \\ g & -\infty & 2^\circ & 0^\bullet & 0 \\ h & 0^\circ & 0^\bullet & -\infty & 2 \\ \hline d_j & 2 & 2 & 0 & \end{array} \tag{9}$$

For this system, (6) gives the system Jacobian

$$\mathbf{J} = \begin{bmatrix} \partial f/\partial x'' & 0 & \partial f/\partial \lambda \\ 0 & \partial g/\partial y'' & \partial g/\partial \lambda \\ \partial h/\partial x & \partial h/\partial y & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 2x & 2y & 0 \end{bmatrix}.$$

From (7) the Taylor index is 3, agreeing with the differentiation index.

2.3 Consistent points and quasi-linearity

The offsets specify what data is needed to define a consistent point, and what consistent means. Namely, the data comprise a point

$$\mathbf{X} = \left(x_1, x_1', \ldots, x_1^{(d_1-1)};\ x_2, x_2', \ldots, x_2^{(d_2-1)}\ ;\ \ldots\ ;\ x_n, x_n', \ldots, x_n^{(d_n-1)} \right). \tag{10}$$

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

It is consistent iff it satisfies the set of equations

$$\mathbf{F} = \left(f_1, f_1', \ldots, f_1^{(c_1-1)}; \; f_2, f_2', \ldots, f_2^{(c_2-1)} \; ; \; \ldots \; ; \; f_n, f_n', \ldots, f_n^{(c_n-1)} \right) = \mathbf{0}. \tag{11}$$

An x_j whose d_j is zero (it must have $\sigma_{ij} = 0$ for all i and thus is a "purely algebraic variable") does not appear in the vector $\mathbf{X}$. Similarly, an f_i with $c_i = 0$ does not appear in $\mathbf{F}$.

f_i' means $\mathrm{d}f_i/\mathrm{d}t$ treating the variables and their derivatives as (unknown) functions of t, for instance if $f_1 = x_1'' - x_1 x_3$ then $f_1' = x_1''' - x_1' x_3 - x_1 x_3'$; similarly for higher derivatives.

A convenient notation for such "irregular vectors" is as follows. If J is a set of indices (j, r) where $1 \le j \le n$ and $r \ge 0$, let x_J denote the set of derivatives $x_j^{(r)}$ for $(j, r) \in J$, regarded as a vector. Use the notation $J_{\le k}$ [resp. $J_{<k}$] to mean the set of (j, r) satisfying $0 \le r \le k + d_j$ [resp. $0 \le r < k + d_j$]. For a similar set of indices I, let f_I denote the derivatives $f_i^{(r)}$ for $(i, r) \in I$, and let $I_{\le k}$ [resp. $I_{<k}$] mean the set of (i, r) satisfying $0 \le r \le k + c_i$ [resp. $0 \le r < k + c_i$].

Then (10, 11) can be written

$$f_{I_{<0}}\left(x_{J_{<0}}\right) = 0. \tag{12}$$

There is an amendment to the above definition of $\mathbf{X}$ and $\mathbf{F}$. DAETS checks whether the next derivatives after those in (10), namely $x_j^{(d_j)}$, $j = 1, \ldots, n$, occur in a *jointly linear* way in the f_i. If they do, we call the DAE *quasi-linear*, by analogy with a similar notion in PDE theory.

If the DAE is *not* quasi-linear, these next derivatives $x_j^{(d_j)}$ are included in $\mathbf{X}$ and corresponding derivatives of the f_i, namely $f_i^{(c_i)} = 0$, $i = 1, \ldots, n$, are included in the equations (11) that define a consistent point. That is, (12) changes to

$$f_{I_{\le 0}}\left(x_{J_{\le 0}}\right) = 0. \tag{13}$$

Define $\alpha = -1$ if the DAE is quasi-linear, and $\alpha = 0$ otherwise. Then write both (12) and (13) as

$$f_{I_{\le \alpha}}\left(x_{J_{\le \alpha}}\right) = 0. \tag{14}$$

For example, in the Pendulum system (8), the relevant derivatives $x_j^{(d_j)}$ are x'', y'' and λ. Their occurrence is jointly linear, so (8) is quasi-linear. It would still be so were, say, f changed to $x''x + x\lambda$ or to $x''y' + x\lambda$, but not if it were changed to $x''y'' + x\lambda$, or to $x''\lambda + x\lambda$.

In the original quasi-linear case, the equations (12) for a consistent point are

$$\text{Solve } h, h' = 0 \;\; \text{for} \;\; x, x', y, y'.$$

When not quasi-linear, they become

$$\text{Solve } f, g, h, h', h'' = 0 \;\; \text{for} \;\; x, x', x'', y, y', y'', \lambda.$$

The *consistent manifold* $\mathcal{M}$ is the solution set of (14) in $x_{J_{\le \alpha}}$ space, regarding all the derivatives as separate independent variables. E.g. for the quasi-linear pendulum system, $\mathcal{M}$ is the set in four-dimensional (x, y, x', y') space where $x^2 + y^2 = L^2$ and $xx' + yy' = 0$.

This is the reason for going from (12) to (13). Consider a particular independent variable value t. Let a set of values $\mathbf{X}$ in (10) be consistent with *some* solution of the DAE at t. If the DAE is quasi-linear, that solution is *unique*. If not, there may be several solutions consistent with these values. However, augmenting $\mathbf{X}$ with the next level of derivatives restores uniqueness.

This affects how the user sets initial conditions. These are usually only guesses of consistent values, which the code corrects by a root-finding process. When the DAE is not quasi-linear, good

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

guesses of the extra derivatives make it more likely that the code finds the consistent point — hence the solution — that was intended.

An example is the use of DAETS to do *arc-length continuation* (see Subsection 5.3). Here t is arc-length from an initial point $\mathbf{x}_0$ along a path $\mathbf{x}(t)$ in some $\mathbb{R}^N$. There is inherent non-uniqueness: from $\mathbf{x}_0$ you can traverse the path in either direction. In this case, the required extra derivatives at a consistent point are the unit tangent $\mathbf{x}_0'$ in the desired direction. That is, the augmented $\mathbf{X}$ is the pair $(\mathbf{x}_0, \mathbf{x}_0')$. A good guess of $\mathbf{x}_0'$ makes the code start off in the right direction.

Quasi-linearity can be used more broadly to reduce the number of initial conditions that need be provided, in a way that is briefly mentioned in Section 5.2.

3 Algorithm overview

We present the stepping algorithm of DAETS and elaborate on various points.

3.1 The stepping algorithm

Initial state: We have an initial value of t and
- Order p of TS expansion;
- Initial solution guess at t comprising $x_{J_{\le\alpha}}$, where α is defined in (14);
- No predicted next step;
- User-supplied point $t_{\rm end}$.

Standard state: We have a current interval $[t_{\rm prev}, t_{\rm cur}]$, of nonzero length, and
- p as above;
- Complete TS $x_{{\rm prev},J_{\le p}}$ at $t_{\rm prev}$;
- $h_{\rm trial}$ = predicted next step;
- Essential part, $x_{{\rm cur},J_{\le\alpha}}$ of (accepted) solution at $t_{\rm cur}$;
- Error estimate e of above accepted solution; necessarily $\|e\| \le$ localtol.
- User-supplied point $t_{\rm end}$; note $t_{\rm end} = t_{\rm prev}$ is allowed.

Algorithm 3.1 (Stepping algorithm)

while $t_{\rm end}$ is not in $[t_{\rm prev},\ t_{\rm cur}]$
  do // Take a step
    if $h_{\rm trial}$ too small **return** htoosmall
    $t_{\rm trial} \leftarrow t_{\rm prev} + h_{\rm trial}$
    // Compute scaled TCs to required order p:
    $x_{{\rm cur},J_{\le p}} \leftarrow$ ComputeTCs($t_{\rm cur}, x_{{\rm cur},J_{\le\alpha}}$, step $= h_{\rm trial}$, order $= p$)
    // Unprojected Taylor series solution and error estimate:
    $[x_{{\rm TS},J_{\le\alpha}},\ e_{\rm TS}] \leftarrow$ SumTS($t_{\rm cur}$, $x_{{\rm cur},J_{\le p}}$, step $= h_{\rm trial}$, order $= p$)
    // Projected Taylor series solution onto the constraints:
    $x_{{\rm trial},J_{\le\alpha}} \leftarrow$ Project($x_{{\rm TS},J_{\le\alpha}}$)
    if (Projection failure) **return** badprojfailure
    // Summary error estimate:
    $e \leftarrow \|e_{\rm TS}\| + \|x_{{\rm trial},J_{\le\alpha}} - x_{{\rm TS},J_{\le\alpha}}\|$
    localtol $\leftarrow$ atol $+ \|x_{\rm trial}\| \times$ rtol
    $h_{\rm trial} \leftarrow$ new predicted value based on e, localtol, p
  until $e \le$ localtol

// Accept the step:
$t_{\rm prev} \leftarrow t_{\rm cur}$, $\quad t_{\rm cur} \leftarrow t_{\rm trial}$
$x_{{\rm prev},J_{\le p}} \leftarrow x_{{\rm cur},J_{\le p}}$, $\quad x_{{\rm cur},J_{\le\alpha}} \leftarrow x_{{\rm trial},J_{\le\alpha}}$

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

if (one-step mode) **return** onestepping
end while

// Now t_{end} is inside $[t_{\mathsf{prev}}, t_{\mathsf{cur}}]$. Compute solution at t_{end}:
$x_{\mathsf{end},J_{\leq\alpha}} \leftarrow \mathsf{SumTS}(t_{\mathsf{prev}},\ x_{\mathsf{prev}},\ \mathsf{step} = t_{\mathsf{end}} - t_{\mathsf{prev}},\ \mathsf{order} = p)$
Project it. // We *assume* the error in this is acceptable
return this projected solution

3.2 About the main loop

We speak of "DAETS" doing the things described below. Mostly, they are features of the `integrate` method of the `DAEsolver` class.

Output points. As with most solvers, we do not want closely spaced output points t_{end} to cause inefficiency by reducing the step size. A good way to do this is for the solver to create a (usually piecewise polynomial) $\mathbf{u}(t)$ approximating the solution to sufficient order, and evaluate this at output points. DAETS does not yet do this. Instead, after stepping to t_i, it remembers the TS expansion at t_{i-1}. If t_{end} is between these two points, it evaluates the TS with a step $t_{\mathrm{end}}-t_{i-1}$, and projects the result to give the output value. Since the error test has already passed with the larger step $h_i = t_i - t_{i-1}$, no error check is done. This handles any number of output points within a step reasonably efficiently, though not quite as well as having a $\mathbf{u}(t)$.

Changing direction. One may have a sequence of t_{end}'s that are not in monotone order, e.g. during the root-finding involved in event location. If they are all in the current interval between t_{i-1} and t_i, the method in the last paragraph handles them with no problem. However, if t_{end} lies outside this interval, on the t_{i-1} side, the integration must change direction. The stored TS at t_{i-1} is re-used, with a step size obtained from stored data, to step to a "new t_i" in the reverse direction. Steps are taken until the new t_{end} is passed, and output produced as in the last paragraph.

One-step mode. When called in one-step mode, DAETS returns at the end of each step, as well as at output points. This can be used to produce output for graphing. It is also used for event location, which at present is not provided by DAETS and must be coded in the calling program.

Initial consistent point. At each step, the algorithm projects a trial solution point onto the consistent manifold $\mathcal{M}$ to give the accepted point. On steps after the first, the step size selection process aims to make the trial point close to $\mathcal{M}$, so a simple projection method suffices. The first step is different in two ways.

First, the task is of *finding a consistent point*, given an initial guess that may be very far from $\mathcal{M}$. This is recognized as one of the challenging problems in solving nonlinear DAEs. The code treats this as a minimisation problem and gives it to the optimisation package IPOPT.

Second, providing the initial values is part of the user interface. It makes sense to split these into *fixed* values, which the user "decides" and wants the code to keep unchanged, and *free* values — "just guesses" that the minimisation process is free to change.

Let the set of values to be found, $x_{J_{\leq\alpha}}$, be regarded as a flat vector $\mathbf{x} = (\mathbf{y}, \mathbf{z})$, where $\mathbf{y}$ and $\mathbf{z}$ denote fixed and free components, respectively. Let the equations by which $\mathcal{M}$ is defined, $f_{I_{\leq\alpha}} = 0$, be denoted $\mathbf{f} = \mathbf{0}$. Let the fixed and free user-supplied values be $\mathbf{y}^*$ and $\mathbf{z}^*$, respectively. Then IPOPT is given the following (generally nonlinear) least-squares problem:

$$\begin{gathered}\min_{(\mathbf{y},\mathbf{z})} \|\mathbf{z} - \mathbf{z}^*\|_2^2 \\ \text{subject to } \mathbf{f}(\mathbf{y},\mathbf{z}) = \mathbf{0} \text{ and } \mathbf{y} = \mathbf{y}^*.\end{gathered} \tag{15}$$

Currently the unweighted 2-norm is used for this task, and for the projections. Because applications often have solution components of very different scales, we aim to offer the option of a weighted norm $\|\mathbf{v}\|^2 = \sum_i w_i |v_i|^2$ in due course.

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

Error tolerance; order and stepsize selection. As with several current ODE/DAE codes, the user (optionally) supplies to DAETS a single tolerance `tol` and says whether this is to be *absolute*, *relative* or *mixed*. These data are translated into the two values `atol` and `rtol` that Algorithm 3.1 uses to compute a local tolerance `localtol` at each step, by the recipe: if absolute, `atol` = `tol`, `rtol` = 0; if relative, `atol` = 0, `rtol` = `tol`; if mixed, `atol` = `rtol` = `tol`. The default is a mixed tolerance of 10^{-12}.

Currently, DAETS uses constant order during an integration, where either DAETS selects a value for the order, or the user can set its value. If the user has not done so, a value is set by

$$p = \lceil -0.5 \ln(\text{tol}) + 1 \rceil, \tag{16}$$

see [12], where $\lceil x \rceil$ denotes the least integer $\geq x$. (Currently a maximum value for the order of the TS is set to 200 at compile time; this can easily be changed to another value.)

The error in an approximate Taylor series solution is estimated as that in the computed approximation to $x_{J_{\leq \alpha}}$. Consider the TS expansion of x_j to order $p + d_j$ at a point t. With stepsize h, DAETS computes scaled TCs at $t + h$:

$$\widetilde{x}_{j,k} \approx x_j^{(k)}(t) h^k / k! \quad \text{for } k = 0, 1, \ldots, p + d_j.$$

Then it computes approximations $\widetilde{x}_j^{(k)}$ to $x_j^{(k)}$ at $t + h$ for all $k = 0, \ldots, d_j + \alpha$:

$$\widetilde{x}_j^{(k)} = \sum_{i=k}^{p+d_j-1} i(i-1)\cdots(i-k+1) \cdot \widetilde{x}_{j,i} / h^k \; + \; r_{j,k},$$

where α is as above, and

$$r_{j,k} = (p + d_j)(p + d_j - 1) \cdots (p + d_j - k + 1) \cdot \widetilde{x}_{j,p+d_j} / h^k.$$

If we denote by r the vector with components $r_{j,k}$ for all $j = 1, \ldots, n$ and all $k = 0, \ldots d_j + \alpha$, we estimate the error in the TS solution by

$$e_{\text{TS}} = \|r\|.$$

We select a trial stepsize, after an accepted or rejected step, by

$$h_{\text{trial}} = h \left(0.25 \cdot \text{tol}/e\right)^{p-\alpha}, \tag{17}$$

where e is computed as in Algorithm 3.1: $e = \|e_{\text{TS}}\| + \|x_{\text{trial},J_{\leq\alpha}} - x_{\text{TS},J_{\leq\alpha}}\|$. In the error estimations, DAETS uses the max norm.

The *initial stepsize* is currently selected as in (17) with $h = 1$. This needs improving to avoid possible overflows/underflows in the TC computation on the first step.

4 The structure of DAETS

DAETS is implemented as a collection of C++ classes. This section describes briefly those classes needed by the user; it outlines the code's internal structure and infrastructural packages; and it lists features in the code and documentation that, because of the role of structural analysis in the solution process, go beyond what is traditional for an ODE solver.

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

4.1 User-visible architecture

The calling program interfaces to DAETS via

- Class `DAEsolver`: this holds integration policy, and data about the DAE.
- Class `DAEsolution`: this holds data about the current solution point.
- Optionally `DAEpoint`: a "cut-down" version of `DAEsolution`.
- Enumerated type `DAEexitflag`: this signals integration success or failure.

The reason for having `DAEsolver` and `DAEsolution` is as follows. An ODE/DAE integration can be viewed as moving a point along a path $\mathbf{x}(t)$ in a space of some dimension m that generally does not equal n and depends on the offsets: see below (18). It is highly desirable to store internal state data — current step size, Jacobian, etc. — in a separate place that "belongs" to the given path. Without this, for instance, it is hard to follow two or more paths in parallel by calling the integrator on each, alternately. For codes written in non-OO languages this storage is typically a work array passed to the integrator. Here, these two classes achieve the separation.

A `DAEsolver` object `Solver` implements the integration process. It contains the needed knowledge about the DAE itself such as the function code and the offsets. It also contains policy data about the integration, such as the Taylor series order, the accuracy tolerance and type of error test (absolute, relative or mixed), and whether the integration is done in one-step mode.

A `DAEsolution` object `X` implements the moving point. This includes the numerical values of its components and the current value of t; also data describing the current state of the solution: are we at an initial guess, a point on the path from which we can integrate further, or a point where an error prevents further progress?

The allocation of attributes between the `DAEsolver` and `DAEsolution` classes is to some extent arbitrary and makes some operations more convenient than others. To use one `Solver` to advance two solutions with different initial conditions, using the same order and tolerance, is easy. To use the same initial conditions but different tolerances, for instance, is possible but less convenient. We could have made the tolerance part of `X` rather than part of `Solver`, but this felt less natural.

This design supports various protective interlocks. A newly created `X` has its `first_entry` flag set true, indicating it is expected to be an inconsistent point. Its t and each component of its `x` is flagged "uninitialized": `integrate` will not accept it until all these values are set. At subsequent consistent points, `first_entry` becomes false and can only be reset by an explicit call to `setFirstEntry()`. When it is false, altering the `x` or t values in `X` is an error. Once integration of `X` has failed, say with "*h too small*", re-calling the integrator raises an error unless one has reset `first_entry`. Such protections are difficult in a traditional code that uses a work array.

The numerical solution values held in `X` are not a flat vector as they are with an ODE, because of the offsets. For instance the simple pendulum has three variables x, y and λ with offsets 2, 2 and 0. In this case `X` stores the values $(x, x',\ y, y')$ — no λ values need be carried. If the problem is modified to be not quasi-linear, DAETS recognizes this, see Subsection 2.3, and `X` stores $(x, x', x'',\ y, y', y'', \lambda)$. To store such data an "irregular array" is used such as

	0	1
0	x	x'
1	y	y'
2		

if quasi-linear, or

	0	1	2
0	x	x'	x''
1	y	y'	y''
2	λ		

if not. (18)

The dimension m mentioned above is thus 4 in the first case, or 7 in the second.

A `DAEpoint` object holds such an irregular array. It supports componentwise $+$, $-$, $\times$ and $\div$ between same-shaped arrays, and the 2-norm. `DAEsolution` objects count as `DAEpoint` objects for

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

this purpose. For instance to form the componentwise relative error `relerr` of a `DAEsolution x` compared with a reference solution held in a `DAEpoint x0`, one may write

```
relerr = norm((x-x0)/x0);
```

Setting the t value in `X` (when the "interlocks" allow it) is done by the `setT()` method. Setting variable/derivative values is done by `setX()` — actually a method of `DAEpoint`: for instance `setX(0,1,3.5)` or `setX(0,1,3.5,0)`, in the array (18), sets derivative 1 of variable 0 to 3.5, that is $x' = 3.5$, as a "free" value. To set it as a "fixed" value, do `setX(0,1,3.5,1)`.

4.2 Internal architecture

The DAETS solver builds on:

- FADBAD++ [27] for computing Taylor coefficients and the System Jacobian;
- LAP [11] for finding a Highest Value Transversal;
- IPOPT [28] for computing a consistent initial point; and
- LAPACK [21] for solving linear systems.

There are classes that overload arithmetic operations and standard functions so that executing the DAE function code computes the signature matrix, quasi-linearity data and other structural information, with the help of the LAP code.

There are classes to support computing and accessing TCs, system Jacobian, the constraints in (15) and a TS solution with a given order and stepsize, and to implement these by FADBAD++.

There are classes to support the method for finding a consistent initial point, and the projection done at each subsequent step; and to implement these by IPOPT.

There is a class to support error estimation and stepsize control.

A class holds constants, such as default tolerance and maximum allowed order, that are hard-wired but which a user may change by amending the source code of this class and recompiling.

4.3 Features to help the user

The theory of the signature matrix, offsets and consistency is given in some detail in the User Guide [22] because it is fairly new and may be unfamiliar to most potential users.

The signature tableau, on the lines of that for the Pendulum example shown in (9), offers much insight into the structure of a DAE, so there is a method `printOffsets()` that displays it.

To use DAETS, a user *must* understand that the needed initial values to be fixed or guessed form an irregular array, see Subsection 4.1. The User Guide explains this, and the code offers help: if the correct set of derivatives has not been initialized on first entry to `integrate`, a message is printed indicating just what this set is.

There is a method to display a requested set of variable and derivative values of a `DAEsolution` point — again, useful because of its irregular shape. There are methods to display statistics about the integration (accepted/rejected steps etc.), and to explain the meaning of a `DAEexitflag` value.

5 Numerical results

Subsection 5.1 shows that DAETS is very accurate on four standard test problems from [14], and compares it with the accuracy of DASSL [2] and RADAU [9]. Subsection 5.2 investigates the performance of DAETS on high-index DAEs, where we solve up to index-47 DAEs. Subsection 5.3 shows DAETS is a quite robust continuation code, using it to solve a class of difficult purely algebraic systems by arc length continuation, formulated as an implicit DAE of index 1.

The codes for these experiments, and many others, are part of the DAETS distribution.

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

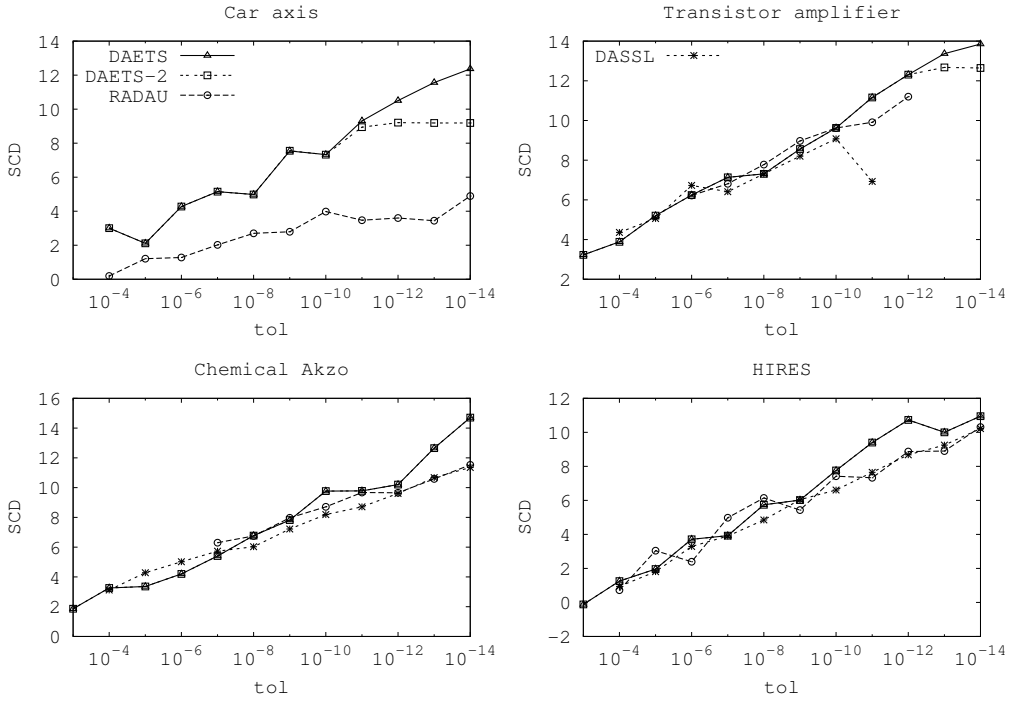

Figure 1: Accuracy of computed solutions by DAETS, RADAU, and DASSL on the car axis, transistor amplifier, chemical Akzo and HIRES problems.

The computations are done on a Mac Pro having two dual-core Intel Xeon, 2.66 GHz processors (four cores in total), 2GB main memory and 4MB L2 cache per processor. DAETS is compiled with the GCC compiler version 4.0.1 with an optimization flag `-O2`; RADAU and DASSL are compiled with G77 version 3.4.2 and optimization flag `-O2`.

The timing results are reported in seconds.

5.1 Accuracy and efficiency

We study the accuracy of the solutions computed by DAETS on four test problems from [14]: *car axis*, an index-3 DAE consisting of 8 differential and 2 algebraic equations; *transistor amplifier*, an index-1 DAE consisting of 8 differential equations; *chemical Akzo Nobel*, an index-1 DAE consisting of 5 differential and 1 algebraic equations; and *HIRES*, an ODE consisting of 8 equations. Except the chemical Azko Nobel problem, these are classified as stiff in [14].

Let a computed value for $x_{J_{\leq\alpha}}$ be regarded as a flat vector $\widetilde{\mathbf{x}}$, and let a reference solution for $x_{J_{\leq\alpha}}$ be regarded as a flat vector $\mathbf{x}_{\text{ref}}$. If $\mathbf{e}$ is the vector with ith component $(\widetilde{\mathbf{x}}_i - \mathbf{x}_{\text{ref},i})/\mathbf{x}_{\text{ref},i}$, we estimate the number of signicant correct digits, SCD, in $\widetilde{\mathbf{x}}$ as in [14]:

$$\text{SCD} = -\log_{10}(\|\mathbf{e}\|_\infty).$$

We integrate these problems with DAETS using a mixed relative–absolute error control with tolerances $\text{tol} = 10^{-4}, 10^{-5}, \ldots, 10^{-14}$. We determine SCD using the reference solutions given in [14] and reference solutions computed by DAETS with $\text{tol} = 10^{-16}$. We also compute SCD with RADAU and DASSL, where we give the same tolerances to these solvers.[4] The latter cannot solve

[4]We give initial stepsize 0 to RADAU, which results in it selecting initial stepsize.

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

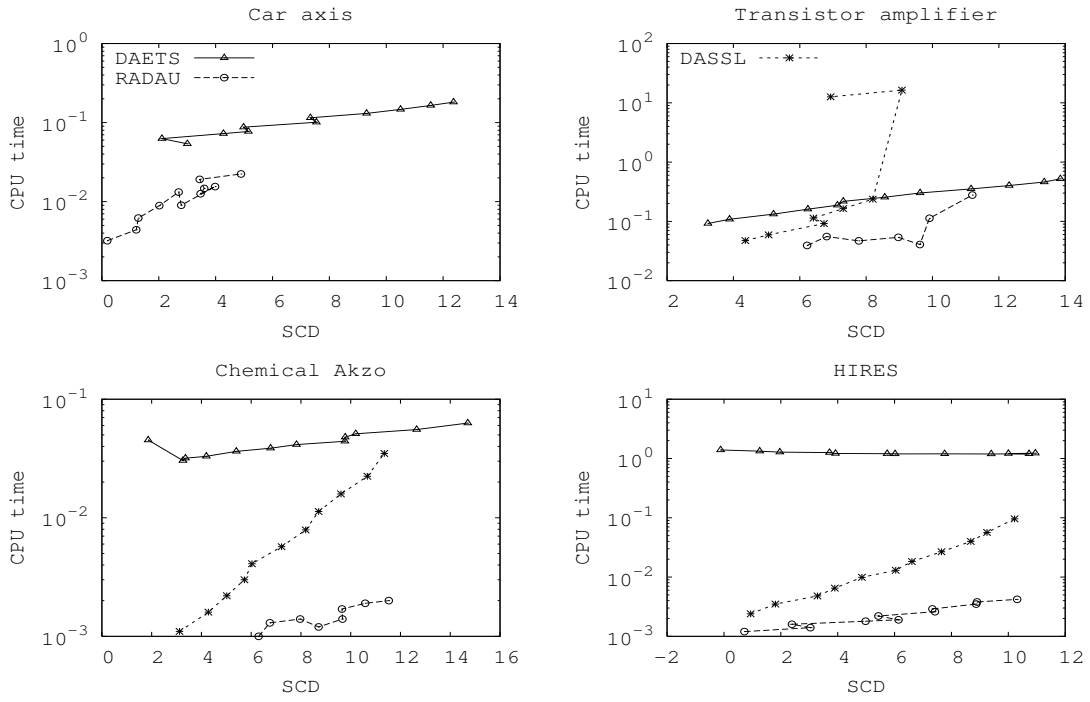

Figure 2: Efficiency of DAETS, RADAU, and DASSL on the car axis, transistor amplifier, chemical Akzo and HIRES problems.

index-3 DAEs, and we do not apply it to the car axis problem.

Plots of SCD versus tol are in Figure 1. Here, "DAETS-2" refers to SCD determined using the reference solutions from [14], and "DAETS " refers to SCD computed using reference solutions computed by DAETS with tol $= 10^{-16}$. These plots show DAETS is highly accurate.

On the transistor amplifier problem, DASSL could not compute a solution with tolerances 10^{-12}, 10^{-13} and 10^{-14}; and RADAU could not compute a solution with tolerances $10^{-4}, 10^{-5}, 10^{-13}$ and 10^{-14}. Also, RADAU could not compute solutions on the chemical Akzo Nobel problem with tolerances 10^{-4}, 10^{-5} and 10^{-6}.

Figure 2 shows work-precision diagrams as described in [14] for DAETS, DASSL, and RADAU on the same four problems. DAETS is not as efficient as standard DAE solvers on problems (and tolerances) for which these solvers perform well. Its strength is in solving high-index problems and in computing accurate solutions at stringent tolerances; cf. Figure 1. For the HIRES problem, the work in DAETS slowly *decreases* as the tolerance decreases. This unusual behaviour is genuine and to do with applying an explicit method of high, tolerance-dependent, order to a stiff problem.

5.2 High-index DAEs

To show how DAETS handles high-index problems we consider P pendula that we imagine to be in a row from left to right:

$$\begin{aligned} 0 &= x_1'' + \lambda_1 x_1 \\ 0 &= y_1'' + \lambda_1 y_1 - G \\ 0 &= x_1^2 + y_1^2 - L^2 \end{aligned} \qquad \text{and} \qquad \begin{aligned} 0 &= x_i'' + \lambda_i x_i \\ 0 &= y_i'' + \lambda_i y_i - G \\ 0 &= x_i^2 + y_i^2 - (L + c\lambda_{i-1})^2, \\ & (i = 2, 3, \ldots, P), \end{aligned} \tag{19}$$

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

where G, L and c are given constants. The first pendulum is undriven; each other in the "chain" receives a driving effect from its left neighbour. The system (19) is of size $3P$ and index $2P+1$. In the numerical experiments that follow, we set $G = 9.8$, $L = 3.4$ and $c = 0.1$.

The theory in Subsection 2.3 says initial conditions are needed for

$$\begin{aligned} &x_i^{(d)},\ y_i^{(d)} \text{ for all } i = 1,\ldots,P \text{ and for all } d = 0,\ldots,2(P-i+1); \\ &\lambda_i^{(d)} \text{ for all } i = 1,\ldots,P-1 \text{ and for all } d = 0,\ldots,2(P-i+1)-1. \end{aligned} \tag{20}$$

That is, earlier pendula in the chain require more ICs. We note below that this can be relaxed.

Exploring chaotic behaviour. To explore how sensitive such chains of pendula are to small changes in initial conditions we create a system comprising two independent chains (19) of size 7. This gives a DAE of size $21 \times 2 = 42$ and index 15. We give initial conditions to the first chain as

$$x_1 = 1, \quad x_1' = 0, \quad y_1 = 0, \quad y_1' = 1, \tag{21}$$

and initialize the other required values in (20) by pseudo-random numbers in $[0, 1)$. The initial conditions for the second chain are those for the first with a small perturbation — each entry multiplied by $1 + 10^{-9}$. The system is integrated from $t_0 = 0$ to $t_{\text{end}} = 60$ with order 30.

Figure 3 shows, above, the computed $x(t)$ for pendulum 7 from the first and second sets of pendula, using tolerance 10^{-9}. Below, it shows the growth in the max norm of the component-wise difference in the solutions from the first and second sets, at tolerances 5×10^{-9} and 10^{-9}. These graphs show the two solutions are nearly identical until around $t = 47$, and then rapidly become apparently quite unrelated — characteristic of chaotic behavior.

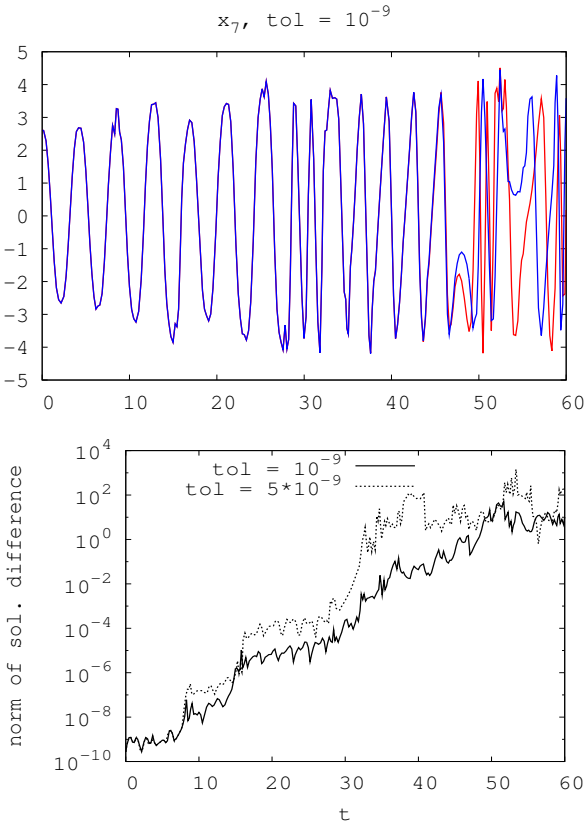

Figure 3: Above: N pendula, deviation of nearby solutions, $N = 7$, tolerance 10^{-9}. Below: Norm of difference in solutions, at two tolerances.

Note on initial conditions. This system illustrates the potential of a deeper analysis of the DAE's structure. We saw the theory in Subsection 2.3 implies earlier pendula in the chain require more ICs. However, in view of the dependencies, it makes no sense that later pendula, of which pendulum 1, say, is "unaware", make it need more ICs. In fact, exploiting quasi-linearity plus the internal block structure of each solution stage, one can reduce the needed ICs to just (x_i, y_i, x_i', y_i') for $i = 1,\ldots,P$. This is a drastic simplification, which also brings the algorithm in line with the physics.

The key tool is the *Dulmage–Mendelsohn* (DM) re-ordering of a sparse matrix to block-triangular form [6, 20]. Bunus and Fritzon [3] report how DM analysis has been used in the Modelica simulation system. We have a prototype DM analysis module, to be included in DAETS in due course.

© 2008 *European Society of Computational Methods in Sciences and Engineering (ESCMSE)*

5.3 A continuation problem

A continuation problem can be viewed as a system of n purely algebraic equations in $n+1$ variables, $\mathbf{f}(\mathbf{y}) = \mathbf{0}$. Generically, the solution is a collection of one-dimensional curves (*branches*). The task is to follow a branch from an initial point, in one of the two possible directions, to a desired point.

A typical practical difficulty is "tracking failure": another branch may come so close that one jumps on to it by mistake. A canonical example (in x, y space, with $n = 1$) is $y^2 - x^2 = c$ where c is a small nonzero constant. If one starts in the third quadrant, say near $(-1, -1)$, and continues toward the origin, one should end up in quadrant 4 if $c > 0$ or quadrant 2 if $c < 0$. However, at fixed precision and for small enough c, any numerical method is bound to "miss" the sharp bend near the origin and end up in quadrant 1. The high-order method used by DAETS makes it rather robust because it enables it to "see" such bends from quite far off.

Continuation is commonly used to solve hard systems of n equations in n variables, using a parameter to move from a trivial problem to the desired one. We illustrate by the "LW problem", from Layne Watson [29]. One seeks a fix-point, that is a root of

$$\mathbf{x} = \mathbf{g}(\mathbf{x}) \tag{22}$$

for the function $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$ where

$$g_i(\mathbf{x}) = g_i(x_1, \ldots, x_n) = \exp\bigl(\cos\bigl(i \sum_{k=1}^{n} x_k\bigr)\bigr), \quad i = 1, \ldots, n. \tag{23}$$

Define $\mathbf{y} = (\lambda, \mathbf{x})$ and consider the parameterized problem

$$\mathbf{0} = \mathbf{f}(\mathbf{y}) = \mathbf{x} - \lambda \mathbf{g}(\mathbf{x}). \tag{24}$$

The idea is to follow the branch through the trivial solution at $\mathbf{y} = (\lambda, \mathbf{x}) = (0, \mathbf{0})$ in the direction of increasing λ until we reach a point with $\lambda = 1$, which will solve our problem.

Even for n as small as 10, solving (22) without continuation is not easy, partly because it has many solutions. Continuation helps; one benefit is that (24) identifies a specific solution, the unique first one on this continuation path. However, even for small n, tracking problems are quite severe.

It is well known that taking λ as the independent variable in such problems generally fails because of *turning points* where $\mathrm{d}\mathbf{x}/\mathrm{d}\lambda$ is infinite. Instead, we use *arc length continuation*, introducing an independent variable s and adding to (24) an $(n+1)$th equation $\left(\|\mathrm{d}\mathbf{y}/\mathrm{d}s\|_2\right)^2 = 1$ that defines s as Euclidean arc-length along the path. That is $0 = S = \sum_j y_j'^2 - 1$, where $'$ means $\mathrm{d}/\mathrm{d}s$.

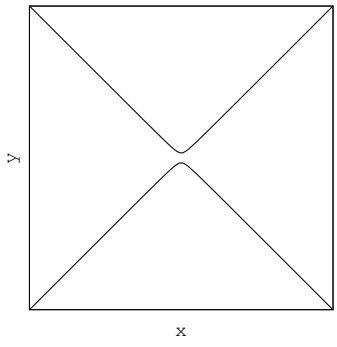

Figure 4: Typical path shape that causes tracking failure.

This formulation gives an index 1 DAE. How does DAETS know which direction to take along the path? The code recognizes the DAE as not *quasi-linear* and thus requires values $x_{J_{\le 0}}$, in the notation of Subsection 2.3, as an initial guess. It turns out that these values comprise $\mathbf{y}$ and $\mathbf{y}'$ — both an initial position, and an initial direction, just what is needed.

Since DAETS currently lacks event location, finding s where $\lambda(s) = 1$ was done by putting the code into one-step mode and, after a step where a sign-change of $(\lambda - 1)$ was observed by the calling program, root-finding by a reverse-communication version of Brent's ZEROIN. Figure 5 shows, for $n = 10$, the graphs of two components x_1 and x_{10} against λ. Many turning points are visible. The path in full $(\lambda, x_1, \ldots, x_n)$ space is convoluted and gets rapidly more so as n increases, as shown by the increasing arc length traversed as λ goes from 0 to 1.

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

Our numerical experiments are about how *robust* DAETS is on the LW problem as n increases. (i) Does it manage to reach $\lambda = 1$? (ii) Does it avoid tracking failures?

We did this by solving, for each n used, with a variety of tolerances and orders. Having no reference solutions, we compared all the vectors $\mathbf{x}$, from those integrations that reached $\lambda = 1$. If there were several such, and they were all essentially equal, we counted them as the correct solution. If not, tracking failure had certainly occurred.

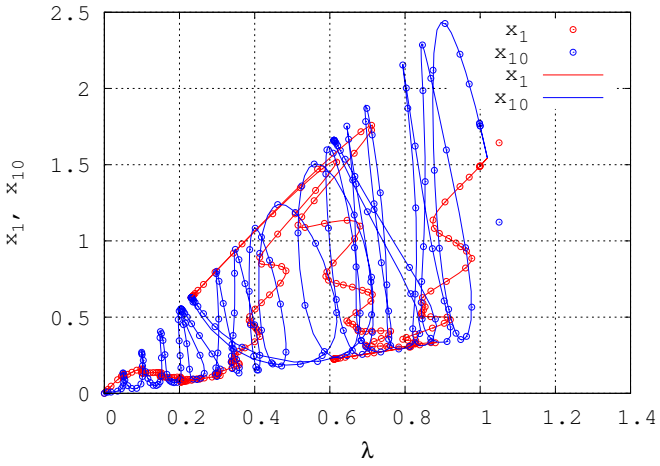

Figure 5: LW problem, $n = 10$. Paths of x_1 and x_{10} plotted against continuation parameter λ. The markers show successive points computed at the slackest tolerance 0.03 at which DAETS succeeded. For the curve, tolerance 1e–7 was used. Points beyond $\lambda = 1$ are part of the event location.

The first tests used orders $p = 0$ (meaning the code's default order formula), 20, 30, 40, 50 and tolerances $\tau = 10^{-3}$, 10^{-5}, 10^{-7}, 10^{-9}, 10^{-11}, 10^{-12} with a mixed error test: 30 runs for each n. When there was *no upper limit* on the stepsize h (that is, it was chosen as described in Section 3), tracking failures were seen for quite small n. Three solutions were found for $n = 4, 6, 9$; four for $n = 12$; six for $n = 14$; one for all other n up to 20. Every computed solution was a "good" one in that the residual norm $\|\mathbf{x} - \mathbf{g}(\mathbf{x})\|_\infty$ was under 10^{-11}.

Things improved when we limited the size of h. The constraint $h \le 0.3$ sufficed to ensure that only one solution was found for each n we tried. The above 30 combinations (p, τ) were used for $n = 1, 2, \ldots, 20$: just two runs failed "*h too small*" at the smallest tolerance. For larger n, fewer (p, τ) were used because of the extensive running times; for $n = 25, 30, 35, 40$ we found only one solution, though more failures at the smallest, and for a reason still obscure, the largest tolerance.

Since the un-limited h could vary from around 10^{-8} to around 6, we guess that the paths have long "easy" stretches, followed by a sudden bend that the code misses if h is un-limited. Our results suggest that the risk of tracking failure on the LW problem depends on geometrical features specific to each n, and does not increase steadily with n.

With the h-limit, which makes little difference to overall run times, the evidence is that DAETS, used as described, is a robust continuation code.

6 A complete code example

Here is code, self-explanatory in the light of the foregoing, for integrating the one-pendulum example (8). We omit output. More extensive examples, with output, are in the User Guide [22].

```
#include "DAEsolver.h"
template <typename T> void fcn(int n,  T t, const T *z, T *f, void *p) {
  // z[0], z[1], z[2] are x, y, lambda.
  const double g = 9.8, L = 10.0;
  f[0] = Diff(z[0],2) + z[0]*z[2];
  f[1] = Diff(z[1],2) + z[1]*z[2] - g;
  f[2] = sqr (z[0])   + sqr(z[1]) - sqr(L);
}
```

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

```
int main() {
  const int n = 3;                          // size of the problem
  DAEsolver Solver(n, DAE_FCN(fcn));        // create a solver, analyse DAE
  Solver.printDAEinfo();                    // print info about the DAE
  Solver.printDAEpointStructure();          // .. and more info
  DAEsolution x(Solver);                    // create a DAE solution object
  x.setT(0.0);                              // initial value of t
  x.setX(0, 0,-1.0).setX(0, 1, 0.0);        // .. and of x and x'
  x.setX(1, 0, 0.0).setX(1, 1, 1.0);        // .. and of y and y'
  double tend = 100.0;
  DAEexitflag flag;
  Solver.integrate(x, tend, flag);          // integrate until tend
  if (flag!=success) printDAEexitflag(flag);// check the exit flag
  x.printSolution();                        // print solution
  x.printStats();                           // print integration statistics
  return 0;
}
```

7 Conclusions

DAETS is a DAE solver based on Pryce's structural analysis (SA) and the use of automatic differentiation to expand the solution in Taylor series. The code is available from the authors under licence. We invite people to submit real-life DAE problems for which the DAETS approach should be suitable, to help us further develop the capabilities of the code.

DAETS has shown itself robust in our experiments. It has as good accuracy-to-tolerance proportionality as do the codes DASSL and RADAU, and far better on the index 3 car axis problem (Subsection 5.1). It is especially effective at high accuracies. Its symbolic understanding of the DAE structure lets it handle high index problems (Subsection 5.2), as well as purely algebraic continuation problems (Subsection 5.3) and explicit or implicit ODEs.

The results of the SA can be printed, which is a help to understanding an unfamiliar DAE. A method to find an initial consistent point is built into DAETS, in contrast to most solvers.

DAETS copes well with moderately stiff problems because of the increasing stability regions of Taylor methods as the order increases. It cannot handle very large problems, very stiff problems, and those where SA gets the structure wrong. These pose three rather different challenges:

Large problems. The difficulty is mainly practical: the memory requirement of high-order TS methods and the computational work associated with AD. One can improve this by using sparse linear algebra and by having fewer long vectors active at once while computing Taylor coefficients. This suggests memory management based on a frontal analysis of the computational graph. However, probably no one wants to solve very large problems to the great accuracy that is the main advantage of Taylor methods.

Stiff problems. There is a real need to develop SA-based methods with suitable stability. A natural extension of our approach would be to adapt *Hermite-Obreschkoff* (HO) methods, which are a sort of Taylor series from both ends of the interval at once. For ODEs they are an effective alternative to Taylor methods with the extra advantage of handling stiffness [8, 18]. It is an interesting task in numerical analysis to devise the formulas and data structures to achieve this for DAEs.

Wrong structural analysis. This event is rare in our experience, but will become commoner as SA-based methods are applied to DAEs generated automatically by software for interactive modelling and simulation. It poses another interesting and difficult task, more in computer science than in numerical analysis. In all cases we have seen, we could make SA get the right answer by rearranging the DAE, manually, in an equivalent form but with "better sparsity". What are the principles and methods involved, and can they be at least partially automated?

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)

References

[1] U. M. Ascher, Hongsheng Chin, and Sebastian Reich. Stabilization of DAEs and invariant manifolds. *Numerische Mathematik*, 67(2):131–149, 1994.

[2] K. Brenan, S. Campbell, and L. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, second edition, 1996.

[3] P. Bunus and P. Fritzon. Methods for structural analysis and debugging of Modelica models. In *Proceedings of the 2nd International Modelica Conference*, pages 157–165. Deutsches Zentrum für Luft und Raumfahrt Oberpfaffenhofen, Germany, March 2002.

[4] S. L. Campbell and C. W. Gear. The index of general nonlinear DAEs. *Numerische Mathematik*, 72:173–196, 1995.

[5] Y. F. Chang and George F. Corliss. ATOMFT: Solving ODEs and DAEs using Taylor series. *Comp. Math. Appl.*, 28:209–233, 1994.

[6] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canad. J. Math*, 1958.

[7] A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, June 1996.

[8] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer-Verlag, second edition, 1991.

[9] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential– Algebraic Problems*. Springer Verlag, Berlin, 1991.

[10] T. E. Hull and W. H. Enright. A structure for programs that solve ordinary differential equations. Technical Report 66, Department of Computer Science, University of Toronto, May 1974.

[11] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987. The assignment code is available at `www.magiclogic.com/assignment.html`.

[12] A. Jorba and M. Zou. A software package for the numerical integration of ODEs by means of high-order Taylor methods. Technical report, Department of Mathematics, University of Texas at Austin, TX 78712-1082, USA, 2001.

[13] S. E. Mattsson and G. Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM J. Sci. Comput.*, 14(3):677–692, 1993.

[14] F. Mazzia and F. Iavernaro. Test set for initial value problem solvers. Technical Report 40, Department of Mathematics, University of Bari, Italy, 2003. `http://pitagora.dm.uniba.it/~testset/`.

[15] N. S. Nedialkov and K. R. Jackson. The design and implementation of a validated object-oriented solver for IVPs for ODEs. Technical Report 6, Software Quality Research Laboratory, Department of Computing and Software, McMaster University, Hamilton, Canada, L8S 4K1, 2002.

[16] N. S. Nedialkov and J. D. Pryce. Solving differential-algebraic equations by Taylor series (I): Computing Taylor coefficients. *BIT*, 45:561–591, 2005.

[17] N. S. Nedialkov and J. D. Pryce. Solving differential-algebraic equations by Taylor series (III): The DAETS code. Technical report, Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, L8S 4K1, November 2007.

[18] N. S. Nedialkov. *Computing Rigorous Bounds on the Solution of an Initial Value Problem for an Ordinary Differential Equation*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, M5S 3G4, February 1999.

[19] C. C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM. J. Sci. Stat. Comput.*, 9:213–231, 1988.

[20] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, December 1990.

[21] LAPACK project. LAPACK — Linear Algebra PACKage. `http://www.netlib.org/lapack/`.

© *2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)*

[22] J. D. Pryce and N. S. Nedialkov. DAETS user guide. Technical report, Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, L8S 4K1, 2007.

[23] J. D. Pryce. Solving high-index DAEs by Taylor Series. *Numerical Algorithms*, 19:195–211, 1998.

[24] J. D. Pryce. A simple structural analysis method for DAEs. *BIT*, 41(2):364–394, 2001.

[25] G. J. Reid, P. Lin, and A. D. Wittkopf. Differential-elimination completion algorithms for DAE and PDAE. *Studies in Applied Mathematics*, 106(1):1–45, December 2001.

[26] G. Reissig, W. S. Martinson, and P. I. Barton. Differential–algebraic equations of index 1 may have an arbitrarily high structural index. *SIAM J. Sci. Comput.*, 21(6):1987–1990, 1999.

[27] O. Stauning and C. Bendtsen. FADBAD++ web page, May 2003. FADBAD++ is available at `www.imm.dtu.dk/fadbad.html`.

[28] A. Wächter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2002.

[29] L. T. Watson. A globally convergent algorithm for computing fixed points of C^2 maps. *Appl. Math. Comput.*, 5:297–311, 1979.

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)