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

Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)
vol. 1, no. 1, 2007, pp. 1-30
ISSN 1790–8140

Solving Differential-Algebraic Equations by Taylor Series (III): the DAETS Code

Nedialko S. Nedialkov [1]

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

John D. Pryce [2]

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

Received —; accepted in revised form —

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++.

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]Corresponding author. E-mail: nedialk@mcmaster.ca
[2]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

$$\frac{(x_1'')^2}{(c^2+(x_1')^2)^3} + t^2 \cos x_2 = 0 \tag{2}$$

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 below.

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. An index of 3 and above is normally considered hard. DAETS is not inherently affected by high index, for reasons explained later. Below, 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 [25], which has proved robust and effective.

Stiff behaviour can be present for DAEs, as for ODEs. DAETS handles moderate stiffness well, but is unsuitable for highly stiff problems.

DAETS is written in C++. Apart from IPOPT, it makes use of Stauning's automatic differentiation (AD) package FADBAD++ [24] and the C version of Volgenant's LAP [10] 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.

The remainder of the paper is laid out as follows. 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 most important components of the code's class structure — both the user-visible and the internal parts. Section 5 describes and discusses the performance of the code on a variety of problems. Section 6 gives examples of coding up a function defining a DAE system, and a main program to solve the problem with DAETS. Section 7 summarises our experience with DAETS and describes some outstanding problems and planned enhancements.

1.2 Background

Many problems of importance in science and industry are modelled as DAEs. Examples from applications are in, for instance, the Test Set for Initial Value Solvers [13], which contains problems from electronic circuits, mechanical systems and chemical processing. Adding detail to a model, such as accounting for the internal behaviour of actuating motors in a robot arm, often increases the DAE 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 [22] based on differential geometry, Campbell's *derivative array* theory [3] and more practical approaches of *index reduction* [1, 12]. DAETS is based on Pryce's *structural analysis* [21], which has this same basic approach and is itself an extension of the method of Pantelides [17].

The software architecture of DAETS is derived from that of Nedialkov's VNODE code [14] 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 [9]. 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) [20, 21], 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 $t = \hat{t}$, it solves implicitly for derivatives, or equivalently TCs, of the solution components $x_j(t)$ at $\hat{t}$.

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 [5], 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 needed 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* $\mathbf{J}$ (4) 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 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 [15, 20, 21].

An SA-based method derives a *structural index* ν_s, which is the same as the index found by the method of Pantelides [17] for DAEs to which that method applies. It is shown by Reißig, Martinson and Barton [23] that ν_s can be arbitrarily greater than ν_d; but in [21, 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 [15] 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 [15, 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 provides the user with much useful information about the structure of the DAE. We provide a way for the calling program to print out the signature matrix, offsets and related data after the SA has been done.

In fact the user *needs* to know SA data in order to use an SA-based method effectively. The reason is that the offsets determine the shape of the set of initial values that must be given to the solver (*fixed* values, or *free*, guessed, values). For traditional ODE or DAE solvers the initial values are flat vectors: here they form an irregular array as in (17) on page 12.

Hence 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, though it may need thought about their physical meaning in order 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 transversal (Subsection 2.1) of finite value, indicating an ill-formulated problem. Or it can appear that the SA succeeded, but the resulting Jacobian matrix $\mathbf{J}$ is always singular, indicating that the SA was unable to reveal the true structure of the DAE. Or DAETS 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 flow of information 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.

2 Theory

2.1 Pryce's Structural analysis of DAEs

We present as much of Pryce's structural analysis [21] as is needed for this paper.

The following definitions are needed. 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. That is, T is 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. The value of a HVT 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 there exists at least one transversal all of whose σ_{ij} are finite. Otherwise the DAE is *structurally ill-posed* — there is probably some error in problem formulation.

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 [21, 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 $\operatorname{Val} \Sigma$:

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

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

   $\mathbf{c}$ 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 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 [21], (5) has the equivalent reformulations:

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

5. Seek values for the x_j and for appropriate derivatives, consistent with the DAE in the sense of Subsection 2.3, and at which $\mathbf{J}$ is nonsingular. If such values are found, 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 [10]. 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

Throughout this paper, we give examples based on the simple pendulum, a DAE of differentiation-index 3. Though this system 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 will be shown by a "tableau", which annotates it with the offsets c_i, d_j and the names of the functions and variables, 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 $\mathbf{d} = (2, 2, 0)$:

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

where a dash denotes $-\infty$. For this system, (6) then 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

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

and 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

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

If we denote

$$\alpha = \begin{cases} -1 & \text{if the DAE is quasi-linear and} \\ 0 & \text{otherwise,} \end{cases}$$

then we write (12) and (13) as

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

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 reason for going from (12) to (13) is as follows. 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. Then if the DAE is quasi-linear that solution is *unique*. If it is not quasi-linear, 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 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 5.6). Here t is arc-length from an initial point $\mathbf{x}_0$ along a path $\mathbf{x}(t)$ in some $\mathbb{R}^m$. 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 $\mathbf{x}_0'$ will make the code start off in the right direction.

3 Algorithm overview

We present the stepping algorithm in DAETS and then elaborate on various points of this algorithm, including order and stepsize selection.

3.1 The stepping algorithm

Initial state: We have an initial t value and

- TS expansion of order p
- Initial solution guess at t comprising $x_{J_{\le\alpha}}$
- No predicted next step
- User-supplied point t_{end}

Standard state: We have a current interval $[t_{\text{prev}}, t_{\text{cur}}]$, of nonzero length, and

- p as above
- Complete TS $x_{\text{prev},J_{\le p}}$ at t_{prev}
- h_{trial} = predicted next step
- Essential part, $x_{\text{cur},J_{\le\alpha}}$ of (accepted) solution at t_{cur}
- Error estimate e of above accepted solution; necessarily $\|e\| \le \text{tol}$
- User-supplied point t_{end}, assumed to be "in front of" t_{prev}, that is $(t_{\text{end}} - t_{\text{prev}})(t_{\text{cur}} - t_{\text{prev}}) \ge 0$; note $t_{\text{end}} = t_{\text{prev}}$ is allowed

Algorithm 3.1 (Stepping algorithm)

```
while t_end is not in [t_prev, t_cur]
  // Take a step
  do
    if h_trial too small return htoosmall
    t_trial ← t_prev + h_trial
     // Compute scaled TCs to required order p:
    x_{cur,J≤p} ← ComputeTCs(t_cur, x_{cur,J≤α}, step = h_trial, order = p)
    // Unprojected Taylor series solution and error estimate:
    [x_{TS,J≤α}, e_TS] ← SumTS(t_cur, x_{cur,J≤p}, step = h_trial, order = p)
    // Projected Taylor series solution on to the constraints:
    x_{trial,J≤α} ← Project(x_{TS,J≤α})
```

if (Projection failure) **return** badprojfailure
// Summary error estimate:
$e \leftarrow \|e_{\mathrm{TS}}\| + \|x_{\mathsf{trial},J_{\le\alpha}} - x_{\mathsf{TS},J_{\le\alpha}}\|$
$\mathsf{tol} \leftarrow \mathsf{atol} + \|x_{\mathsf{trial}}\| \times \mathsf{rtol}$
$h_{\mathsf{trial}} \leftarrow$ new predicted value based on e, tol, p
while $e >$ tol
// Now accept the step:
$t_{\mathsf{prev}} \leftarrow t_{\mathsf{cur}}, \quad t_{\mathsf{cur}} \leftarrow t_{\mathsf{trial}}$
$x_{\mathsf{prev},J_{\le p}} \leftarrow x_{\mathsf{cur},J_{\le p}}, \quad x_{\mathsf{cur},J_{\le\alpha}} \leftarrow x_{\mathsf{trial},J_{\le\alpha}}$
if (one-step mode) **return** onestepping
end while
// Now t_{end} is inside $[t_{\mathsf{prev}}, t_{\mathsf{cur}}]$, compute solution at t_{end}:
$x_{\mathsf{end},J_{\le\alpha}} \leftarrow \mathsf{SumTS}(t_{\mathsf{prev}},\ x_{\mathsf{prev}},\ \mathsf{step} = t_{\mathsf{end}} - t_{\mathsf{prev}},\ \mathsf{order} = p)$
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.

3.2.1 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, see Section 7. Instead, after stepping to t_i, DAETS 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 $\mathbf{u}(t)$.

3.2.2 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 mechanism in the last paragraph handles them with no problem. If however 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. Further steps are taken as necessary till the new t_{end} is passed, and output produced as in the last paragraph.

3.2.3 One-step mode.

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

3.2.4 Finding an 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, the initial guess is part of the user interface. It makes sense to split its components into *fixed* values, which the user "decides" and wants to keep as is, 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:

$$\begin{gathered} \min_{(\mathbf{y},\mathbf{z})} \|\mathbf{z} - \mathbf{z}^*\|_2^2 \\ \text{subject to } \mathbf{f}(\mathbf{y}, \mathbf{z}) = \mathbf{0}, \\ \text{and } \mathbf{y} = \mathbf{y}^*. \end{gathered} \tag{14}$$

Currently the unweighted 2-norm is used; we intend to provide the option of a weighted 2-norm $\|\mathbf{v}\|^2 = \sum_i w_i |v_i|^2$ in the future.

3.2.5 Order and stepsize selection

Currently, DAETS uses constant order during an integration, where either DAETS selects a value for the order or the user can set a value for it. If the user has not specified such, a value is selected by [11]

$$p = \lceil -0.5 \ln(\text{tol}) + 1 \rceil, \tag{15}$$

where tol is either the default tolerance of 10^{-12} in DAETS or a user-specified value. (Currently a maximum value for the order of the TS is set to 200 at compile time, but this can be easily 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 $t + h$:

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

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

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

where α is as above, and

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

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

$$h_{\text{trial}} = h \left(\frac{0.25 \cdot \text{tol}}{e} \right)^{p-\alpha}, \tag{16}$$

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 (16) with $h = 1$, but in the future we shall incorporate a better strategy that avoids 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 the classes, first those needed by the user and then the internal ones. Only a few methods and attributes are mentioned: for more information see the User Guide [19].

4.1 User-visible architecture

The calling program uses two classes `DAEsolver` and `DAEsolution`; optionally, a "cut-down" version of `DAEsolution` called `DAEpoint`; and an enumerated type `DAEexitflag` that 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 some $\mathbb{R}^m$. 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 that it is expected to be an inconsistent point. Its t, and each component of its

`x`, is flagged as "uninitialized", and `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, trying to alter the `x` or t values in `X` is an error. Similarly, once integration of `X` has failed, say with "step size 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 3 variables x, y, λ with offsets $2, 2, 0$. In this case `X` stores the values of x, x', y, y' — no λ values need be carried. If the problem were modified to be not quasi-linear, DAETS would recognize this and (as explained below) make `X` store up to an extra level of derivatives, that is x, x', x'', y, y', y'', λ. To store such data an "irregular array" is used such as

	0	1
0	x	x'
1	y	y'
2		

or, if not quasi-linear,

	0	1	2
0	x	x'	x''
1	y	y'	y''
2	λ		

. (17)

A `DAEpoint` object merely holds such an irregular array. It supports vector space operations: + and − between same-shaped arrays; scalar multiplication; and the 2-norm. `DAEsolution` objects count as `DAEpoint` objects for this purpose. For instance to check a solution `X` at some t against a reference solution held in a `DAEpoint x0`, one may subtract one from the other, giving a `DAEpoint` result, and take the norm of the result.

Setting the t value in `X` (when allowed) 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 above, 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

Infrastructure. The DAETS solver builds on:

- FADBAD++ [24] for computing Taylor coefficients and the System Jacobian;
- LAP [10] for computing an HVT;
- IPOPT [25] for computing a consistent initial point; and
- LAPACK [18] for solving linear systems.

Classes. We give brief, informal descriptions of the rest of the classes in DAETS and display the related class diagram in Figure 1.

`TaylorSeries` is a pure virtual class whose main purpose it to provide virtual functions for computing TCs, system Jacobian, the constraints in (14) and a TS solution with a given order and stepsize. It also supplies virtual functions for accessing TCs.

`FadbadTS` implements the functionality of `TaylorSeries` using the FADBAD++ package.

`Projection` provides functions for finding a consistent initial point and projecting a TS solution.

`IpoptProj` interfaces the IPOPT package, which is used for computing a consistent initial point.

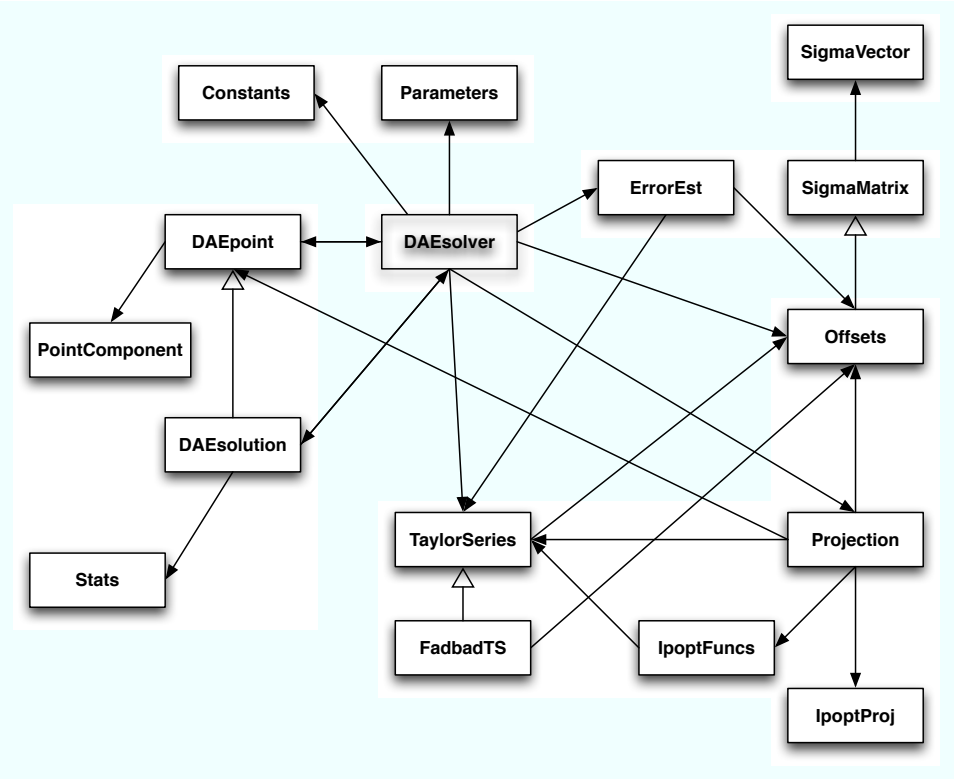

Figure 1: DAETS class diagram. The arrows with the triangle denote inheritance; a "normal" arrow from class A to class B means A uses B.

`IpoptFuncs` provides the functions needed by IPOPT in (14): for evaluating the objective, its gradient, the constraints and the Jacobian of the constraints on each stage.

`SigmaVector` overloads the arithmetic operations and standard functions, including the `Diff` operator, for "sigma vectors"; see also [15].

`SigmaMatrix` provides functions for computing the signature matrix of a DAE and determining if the problem is quasi-linear. These computations are performed by propagating `SigmaVector` objects through the code list of the DAE.

`Offsets` provides functions for computing the offsets of the problem, using its signature matrix.

`Constants` stores various constants, such as default values for order, tolerance, etc. needed during integration.

`Parameters` encapsulates various parameters that can be set before integration, such as order, tolerance, smallest allowed stepsize, etc.

`ErrorEst` estimates the error in a TS soluion.

`PointComponent` describes an entry in a `DAEpoint` object: a `PointComponent` object contains its numerical value, and says whether it is fixed or free and whether it is initialized.

`Stats` collects various statistics during an integration, such as CPU time, number of steps, percentage rejected steps, etc.

4.3 Help for the user

This subsection lists features in the code and documentation that go beyond what is traditional for an ODE solver.

The theory of signature matrix and offsets is given in some detail in the User Guide [19] because it is fairly new and will 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 above. The User Guide explains this in detail, and also 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.

More familiarly, 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 [13]. Subsection 5.2 examines the efficiency of DAETS on these problems and compares it with the efficiency of DASSL [2] and RADAU [7]. Subsection 5.3 investigates the performance of DAETS on high-index DAEs, where we solve up to index-47 DAEs. Subsection 5.4 studies the efficiency of DAETS on the heat equation discretized by the method of lines and formulated as an index-1 DAE. Subsection 5.5 illustrates how the total number of steps behaves as the order of the method increases. Finally, Subsection 5.6 shows that DAETS can solve a pure algebraic system using continuation and formulated as an implicit DAE of index 1.

The computations are performed 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

We study the accuracy of the computed solutions by DAETS on four test problems from [13]: 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, the rest are classified as stiff in [13].)

Let the set of computed values (with a given tolerance) 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 number of signicant correct digits, SCD, in $\widetilde{\mathbf{x}}$ as in [13]:

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

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 [13] 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.[3] The latter cannot solve index-3 DAEs, and we do not apply it to the car axis problem.

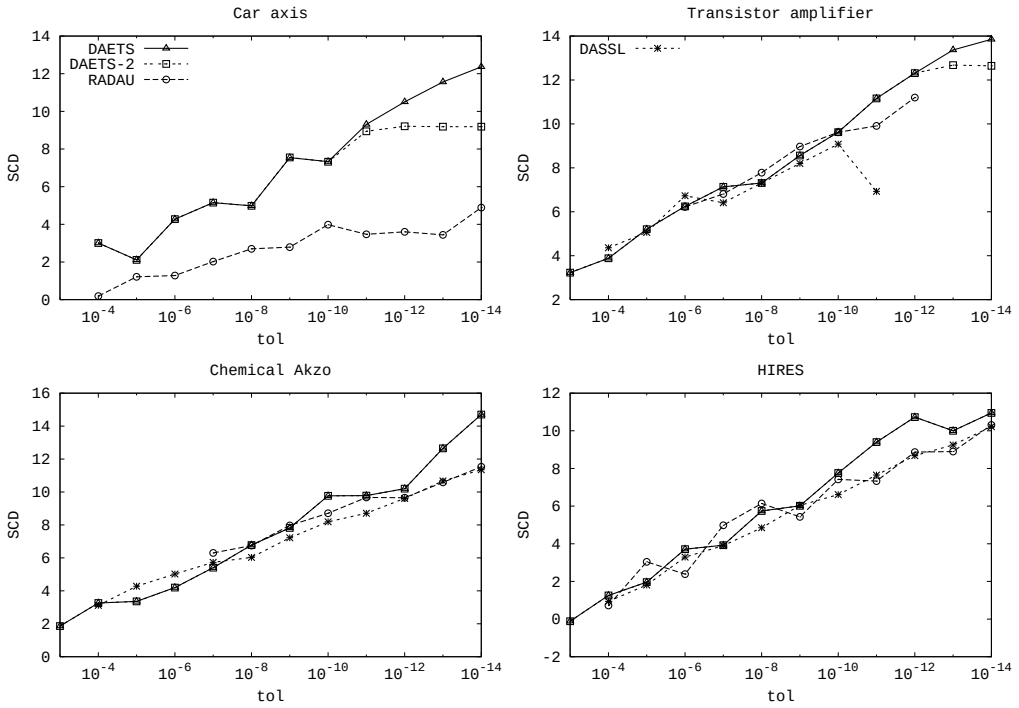

Figure 2: Accuracy of computed solutions by DAETS, RADAU, and DASSL on the car axis, transistor amplifier, chemical Akzo and HIRES problems.

Plots of SCD versus tol are displayed in Figure 2. In this Figure, "DAETS" refers to SCD computed using reference solutions computed by DAETS with tol $= 10^{-16}$, and "DAETS-2" refers to SCD determined using the reference solutions from [13]. It is clear from these plots that 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 for this problem 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}.

5.2 Efficiency

In Figure 3, we plot the work-precision diagrams as described in [13] for DAETS, DASSL and RADAU on the above problems.

On the HIRES problem, the work in DAETS slowly decreases as the tolerance decreases. We believe this is because of an under-appreciated effect when (explicit) methods of high order are applied to stiff ODEs. For many stiff problems, after a rapidly changing "transient phase", solutions $x(t)$ approach a slowly varying "stable solution" $S(t)$. If, at $t = t_r$, S has a well-behaved (large radius of convergence) TS and the computed approximation x_r to $x(t_r)$ is close enough to $S(t_r)$, then the TS at (t_r, x_r) is also well-behaved and approximates $S(t)$ closely even at quite large step

[3]We give initial stepsize 0 to RADAU, which results in the solver selecting initial stepsize.

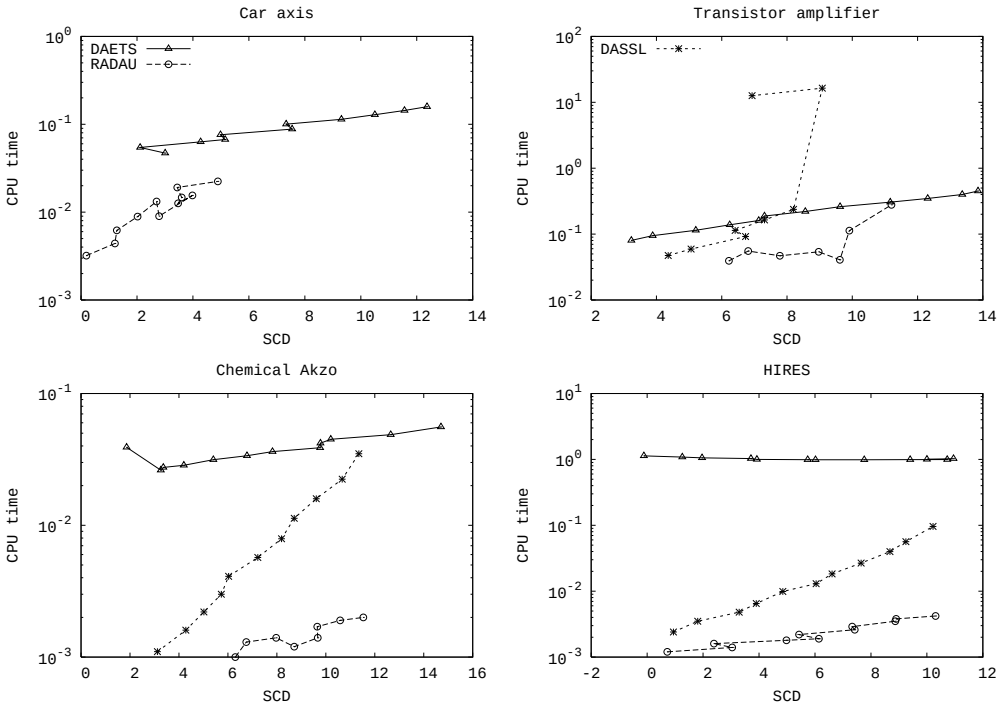

Figure 3: Efficiency of DAETS, RADAU, and DASSL on the car axis, transistor amplifier, chemical Akzo and HIRES problems.

sizes. Thus the local error is small enough not to "re-awaken" transients, and this good behaviour continues at subsequent steps. The higher the order, the stronger is this effect.

We do not yet understand the phenomenon well. It will not apply to all stiff systems; and it probably depends on our step size control algorithm being sufficiently cautious to keep well within the stability region.

Generally, our solver is not as efficient as standard DAE solvers on problems (and tolerances) for which these solvers perform well. The strength of DAETS is in solving high-index problems, which standard solvers cannot tackle, and in computing accurate solutions at stringent tolerances; cf. Figures 2 and 3.

5.3 High-index DAEs

We consider the DAE problem consisting of P pendula:

$$\begin{array}{lll} 0 = x_1'' + \lambda_1 x_1 & & 0 = x_i'' + \lambda_i x_i \\ 0 = y_1'' + \lambda_1 y_1 - G \quad \text{and} & & 0 = y_i'' + \lambda_i y_i - G \qquad (i = 2, 3, \ldots, P), \\ 0 = x_1^2 + y_1^2 - L^2 & & 0 = x_i^2 + y_i^2 - (L + c\lambda_{i-1})^2, \end{array} \tag{18}$$

where G, L and c are given constants. Here, the first pendulum is undriven, and pendulum $(i-1)$ exerts a driving effect on pendulum i, for $i = 2, \ldots, P$. The system (18) is of size $3P$ and index $2P + 1$.

Our solver requires initial conditions for

$$\begin{aligned} x_i^{(d)},\ y_i^{(d)} \quad & \text{for all } i = 1, \ldots, P \quad \text{and for all } d = 0, \ldots, 2(P-i+1); \text{ and} \\ \lambda_i^{(d)} \quad & \text{for all } i = 1, \ldots, P-1 \quad \text{and for all } d = 0, \ldots, 2(P-i+1)-1. \end{aligned} \tag{19}$$

That is, the earlier pendula in the sequence require more initial conditions. In the numerical experiments that follow, we set $G = 9.8$, $L = 3.4$ and $c = 0.1$.

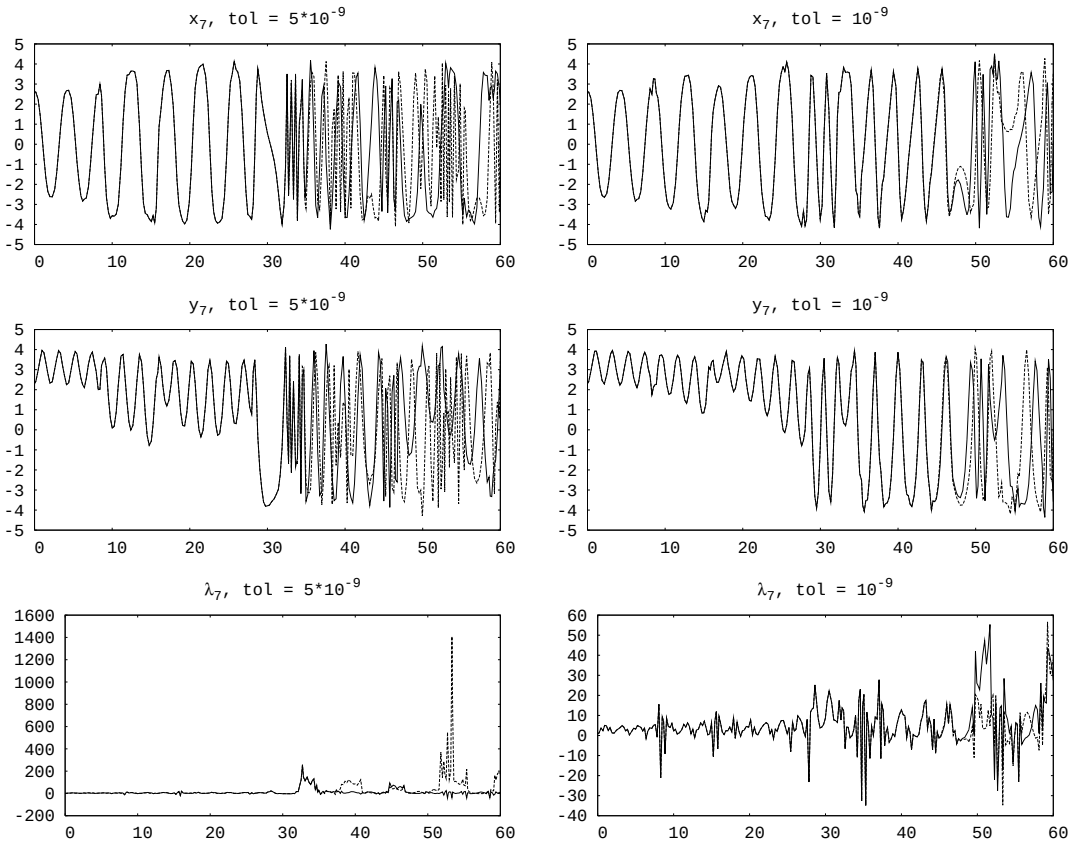

Figure 4: Solutions to pendulum 7 from the first (solid) and second (dashed) sets of pendula computed with tolerances 5×10^{-9} and 10^{-9}.

Solution to index-15 DAE. We integrate two uncoupled sets of pendula each consisting of 7 pendula (18). This results in a DAE system of size 42 and index 15. We give initial conditions to the first set as

$$x_1 = 1, \quad x_1' = 0, \quad y_1 = 0, \quad y_1' = 1, \tag{20}$$

and initialize the rest of the required values in (19) by randomly generated numbers in $[0, 1)$. The initial conditions for the second set are the initial condition for the first one with each entry multiplied by $1 + 10^{-9}$. We use order 30 and integrate from $t_0 = 0$ to $t_{\text{end}} = 60$.

In Figure 4, we plot the computed solutions to pendulum 7 from the first and second sets, where we use tolerances 5×10^{-9} and 10^{-9}. In Figure 5, we plot the max norm of the component-wise

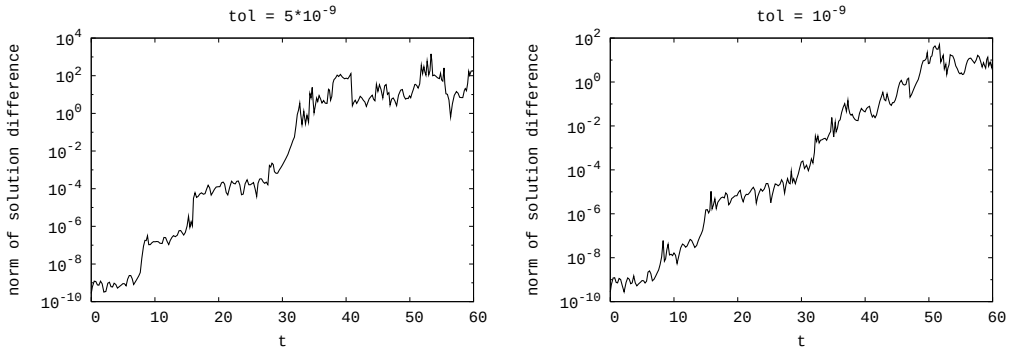

Figure 5: Norm of difference in solution components from first and second sets of pendula.

difference in the solutions from the first and second sets of pendula versus t. Obviously, these solutions exhibit a chaotic behavior.

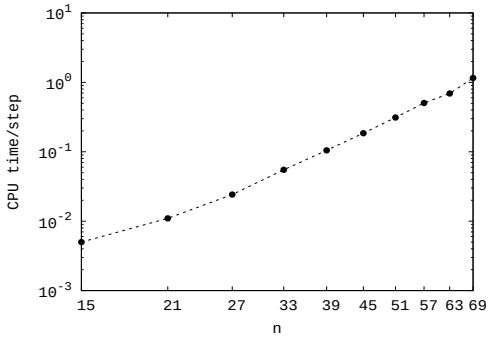

Figure 6: Work per step versus the size of the N-pendula problem (18).

Work per step versus problem size. We integrate (18) for $P = 5, 7, \ldots, 23$, which results in problem sizes $15, 21, \ldots, 69$ and index $11, 15, \ldots, 47$ DAEs. We use order 30 and tol $= 10^{-10}$ and integrate from $t_0 = 0$ to $t_{\text{end}} = 10$. We set initial conditions for the first pendulum as in (20) and set random numbers from $[0, 1)$ for the remaining initial conditions. In Figure 6, we plot the CPU time per step versus the size of the problem. A least squares fit shows that (on this problem) the work per step grows like $\approx n^{3.6}$, where n is the size of the problem.

Work versus order. In Figure 7, we plot the CPU time versus the order of the Taylor series for $P = 1, 5, 9$ and 13 (resulting in index 3, 11, 19 and 27, respectively DAEs) and tolerances $10^{-4}, 10^{-6}, \ldots, 10^{-12}$. The integrations are from $t_0 = 0$ to $t_{\text{end}} = 10$. (Initial conditions are set as in the previous experiment.) On this problem, as the size of the problem increases, the "optimal" order increases as well.

5.4 A DAE from the method of lines

We consider the heat equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2},$$

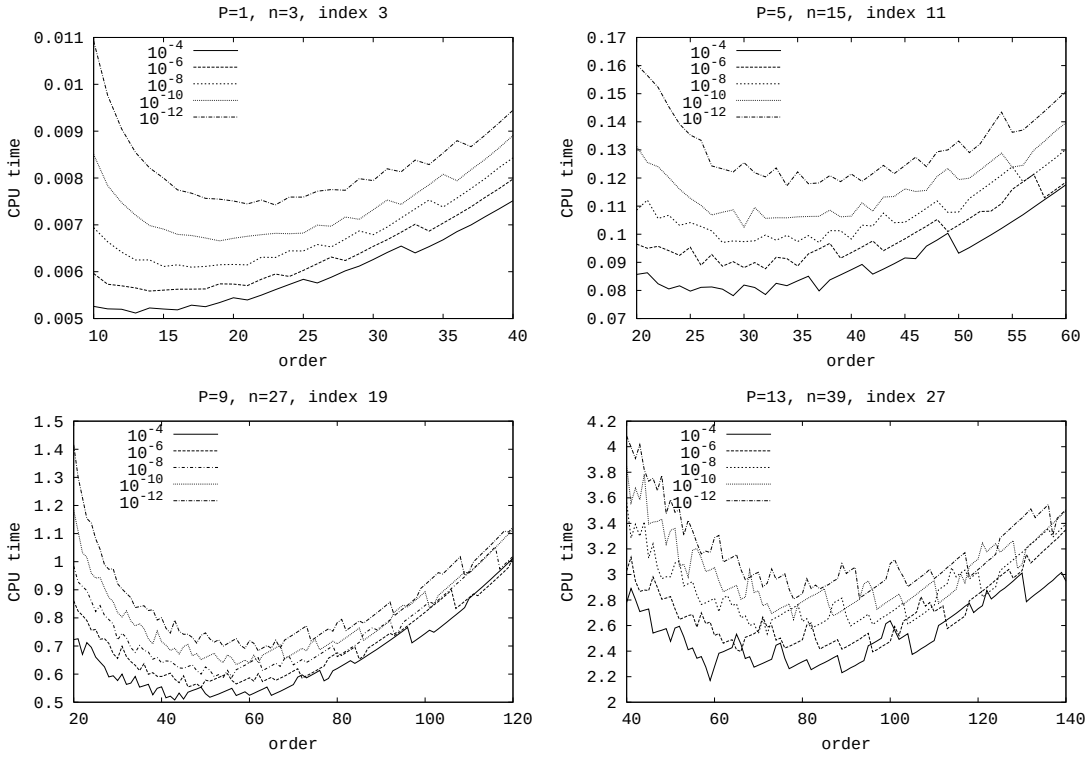

Figure 7: Work versus the order of the method on the N pendula (18).

where $u = u(t, x)$ and D is a real positive constants. We use the initial condition

$$u(0, x) = \sin(x) \quad \text{for all } x \in [0, L],$$

where $L > 0$ is a constant, and boundary conditions

$$u(t, 0) = 1 \quad \text{and} \quad \frac{\partial u}{\partial x}(t, L) = 0 \quad \text{for all } t \in [0, t_{\text{end}}].$$

Setting $h = L/m$, where m is a positive integer, and using second-order central difference approximation to $\partial^2 u/\partial x^2$, we obtain the ODEs

$$\frac{du_i}{dt} = D\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}, \quad i = 1, 2, \ldots, m. \tag{21}$$

We approximate

$$0 = \frac{\partial u}{\partial x} \approx \frac{u_{m+1} - u_{m-1}}{2h} = 0,$$

from which we obtain the algebraic constraint

$$0 = u_{m+1} - u_{m-1}. \tag{22}$$

Combining (21) and (22), we obtain an index-1 DAE system. In the numerical results that follow, we chose $D = 0.96$, $L = 10$.

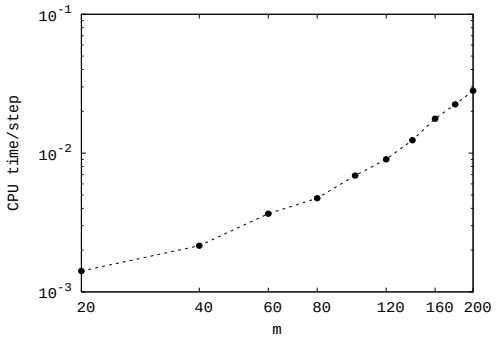

Figure 8: Work per step versus the size of the discretized heat equation.

Work per step versus problem size. In Figure 8, we plot the CPU time per step versus the size of the problem when integrating (21–22) with the above initial and boundary conditions. We use order 30 and tol $= 10^{-10}$ and integrate from $t_0 = 0$ to $t_{\text{end}} = 5$. A least squares fit shows that this work grow like $\approx n^{2.1}$.

The reason DAETS is more efficient on this problem than on the pendula problem from the previous subsection is the amount of automatic differentiation involved. Here, we have a linear problem, and to compute p coefficients we require $O(p)$ work, while to compute TCs for a nonlinear problem, we require $O(p^2)$ work.

Work versus order. In Figure 10, we show the work versus order for tolerances tol $= 10^{-4}, 10^{-6}, \ldots, 10^{-12}$ when integrating (21–22) with the above initial and boundary conditions from $t_0 = 0$ to $t_{\text{end}} = 10$. As the discretization in x becomes finer, the problem becomes stiffer, which results in increasing the "optimal" order as m increases, due to the Taylor method's stability region increasing with order, see Figure 9.

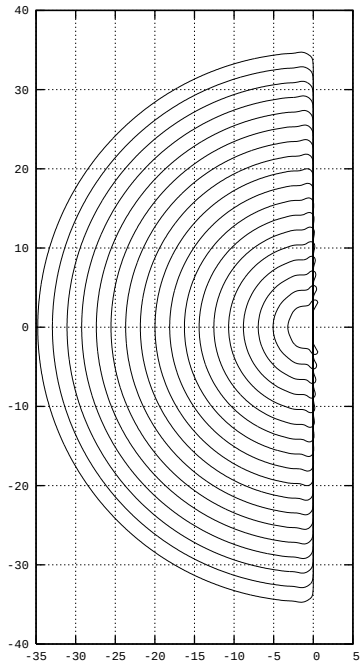

Figure 9: Stability regions of the Taylor series method for orders p =5, 10, 15, ..., 90. They are very nearly semicircles with radius r given by $r \approx 0.375p + 1.352$. The lobes near the y axis seem to get smaller with p but actually have a periodic pattern.

5.5 Number of steps versus order

We illustrate how the total number of integration steps behaves as the order of the integration increases. In Figure 11, we plot number of steps versus the order of the method when integrating the car axis, HIRES, the pendula problem with 5 and 13 pendula, and the discretized heat equation with $m = 30$ and $m = 300$. As expected, when the order increases, this number goes down due to the smaller truncation error and extended stability region of the Taylor series method.

The behavior of the number of steps on the HIRES problem seems unusual: DAETS takes more steps as the tolerance becomes more relaxed from 10^{-12} to 10^{-4}. A possible explanation is along the lines of Subsection 5.2.

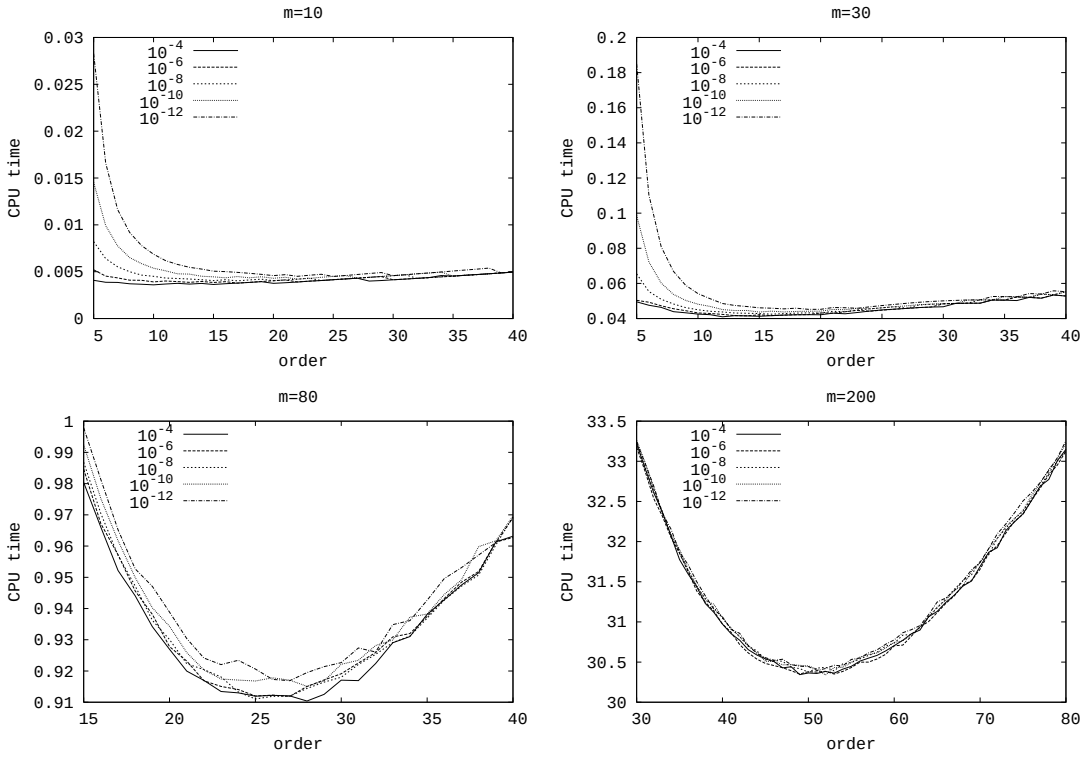

Figure 10: Work versus order in DAETS on the heat equation.

5.6 A continuation problem

This problem comes from Layne Watson [26]. We seek a fix-point, that is a root of $\mathbf{x} = \mathbf{g}(\mathbf{x})$, for the nonlinear function $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$ defined by

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

Even for n as small as 10 this is considered quite difficult. We use the approach of seeking a fix-point of the parameterized problem $\mathbf{x} = \lambda\mathbf{g}(\mathbf{x})$, that is a root of

$$\mathbf{f}(\lambda, \mathbf{x}) = \mathbf{0} \tag{24}$$

where $\mathbf{f}(\lambda, \mathbf{x}) = \mathbf{x} - \lambda\mathbf{g}(\mathbf{x})$. When $\lambda = 0$ it has the trivial solution $\mathbf{x} = \mathbf{0}$, and we hope to track a solution all the way to the desired root at $\lambda = 1$.

DAETS can handle this directly, taking λ as the independent variable and solving for $\mathbf{x} = \mathbf{x}(\lambda)$. This *simple continuation* approach gives an index 0 system. All the offsets c_i and d_j are zero, and the System Jacobian is

$$\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = I - \lambda \frac{\partial \mathbf{g}}{\partial \mathbf{x}}.$$

For general problems (24) this approach works as long as $\mathbf{f}_\mathbf{x} = \partial\mathbf{f}/\partial\mathbf{x}$ is nonsingular, but has a serious weakness: with high probability one passes λ values (reciprocals of eigenvalues of $\mathbf{f}_\mathbf{x}$)

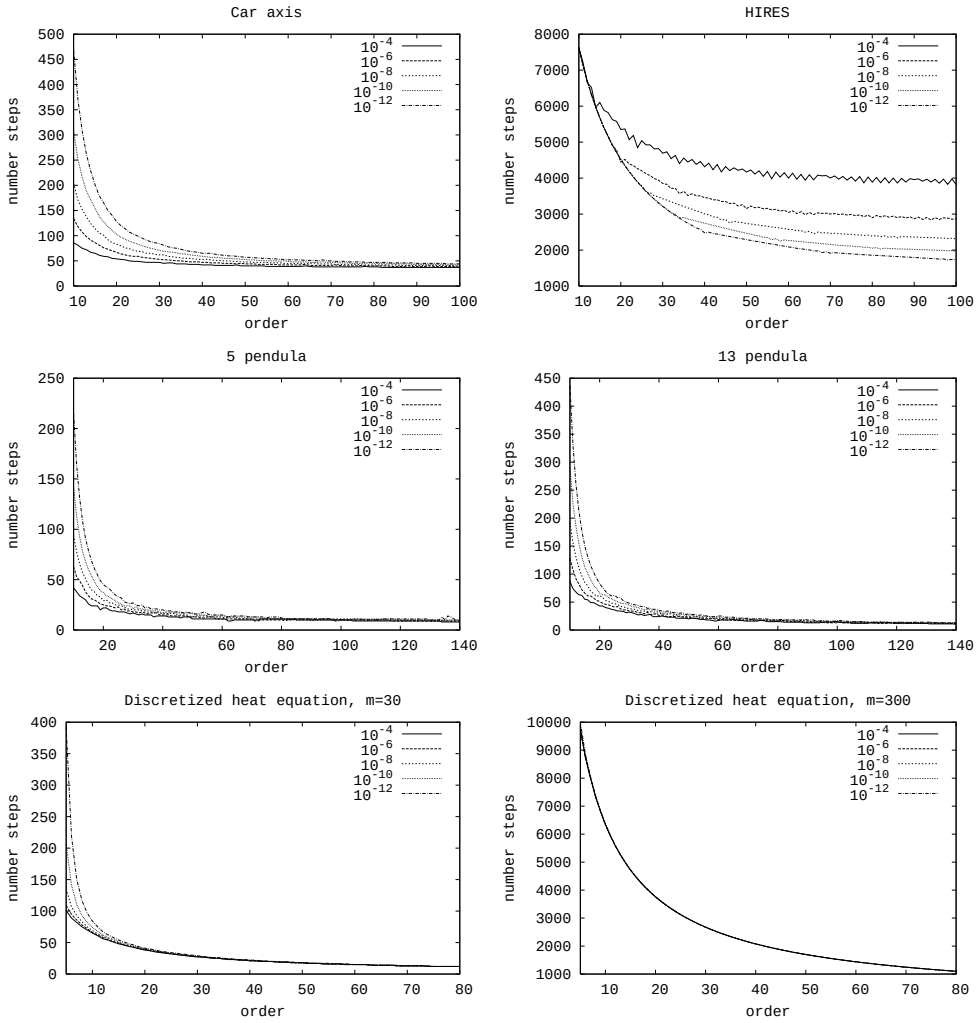

Figure 11: Plots of number of steps versus the order of the method.

where $\mathbf{f_x}$ becomes singular, even though the $n \times (n{+}1)$ matrix $[\mathbf{f}_\lambda,\ \mathbf{f_x}]$ retains full rank n. These are *turning points*, at which $\mathrm{d}\mathbf{x}/\mathrm{d}\lambda$ becomes infinite. For (23), simple continuation only works for $n = 1$ and 2.

A powerful alternative is to treat $\lambda, x_1, \ldots, x_n$ as all on the same footing instead of viewing λ as special, and to use *arc-length continuation*. The system (24) may be rewritten as n equations in $n{+}1$ unknowns:

$$\mathbf{f}(\mathbf{y}) = \mathbf{0} \tag{25}$$

where $\mathbf{y} = (\lambda;\, \mathbf{x})$. As long as $\mathbf{f_y}$ is of full row rank there is a (unique except for direction) solution path through any point in $\mathbb{R}^{n+1}$ satisfying (25), tangential to the 1-dimensional null space of $\mathbf{f_y}$. Generically, i.e. for a "randomly chosen problem", this is true everywhere along the path. Invent

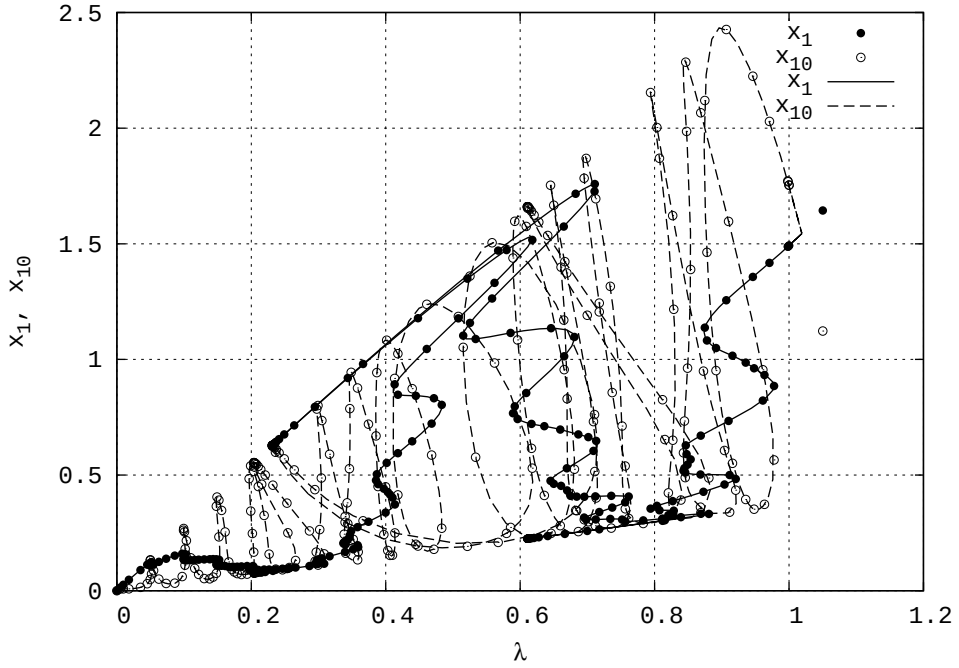

Figure 12: Layne Watson problem, $n = 10$. The paths of x_1 and x_{10} are plotted against the original continuation parameter λ. Many turning points can be seen. The markers show successive points computed at the slackest tolerance 0.03 at which DAETS succeeded. For the curve, tolerance 1e–7 was used. Points beyond $\lambda = 1$ are part of the event location.

a new independent variable s and add to (25) an equation $\|\mathrm{d}\mathbf{y}/\mathrm{d}s\|_2^2 = 1$, that is

$$0 = S = \sum_j {y'_j}^2 - 1, \tag{26}$$

where $'$ means $\mathrm{d}/\mathrm{d}s$. This defines s to be Euclidean arc-length along the path; one can insert weights to scale the components of $\mathbf{y}$ if needed.

This formulation gives an index 1 system of $n + 1$ equations and variables. To match array indexing in C++ we label these from 0 to n with λ being variable 0 and (26) being equation 0. The offsets are $c_0 = 0$, $c_1, \ldots, c_n = 1$ and all $d_0, \ldots, d_n = 1$. The $(n{+}1) \times (n{+}1)$ System Jacobian is

$$\mathbf{J} = \begin{pmatrix} 2(\mathbf{y}')^T \\ \mathbf{f_y} \end{pmatrix},$$

which is nonsingular if $\mathbf{f_y}$ has full rank, since $\mathbf{y}'$ is orthogonal to the rows of $\mathbf{f_y}$.

How does DAETS know which direction to take along the path? The code recognizes the resulting DAE as not *quasi-linear* (Subsection 2.3) and thus requires values $x_{J_{\le 0}}$, in the notation of that subsection, as an initial guess. Since all offsets d_j are 1, these values are $y_1, \ldots, y_{n+1}$; $y'_1, \ldots, y'_{n+1}$ — not only an initial position, but also an initial direction, just what is needed.

Since DAETS currently lacks event location, finding the arclength value s where $\lambda(s) = 1$ was done by the calling program. Stage 1: integrate in one-step mode up to the first s-step for which $\lambda > 1$. Stage 2: do a simple Newton iteration within this step.

A larger tolerance τ_1 was used for stage 1 since the aim is to follow the path quickly; a smaller tolerance $\tau_2 = 10^{-10}$ was used for the final Newton iteration. As n increased τ_1 had to be made smaller, else a projection failure occurred somewhere on the path. Taking $\tau_1 = 10^{-8}$ worked to find a solution for all n we tried, though $\tau_1 = 0.03$ worked for $n = 10$.

Figure 12 shows, for $n = 10$, the graphs of two components x_1 and x_{10} against λ. The values at $\lambda = 1$ are

s	8.7504e+01	λ	1.0000000e+00
x_1	1.4919137e+00	x_2	5.0666536e−01
x_3	3.8904338e−01	x_4	9.2731714e−01
x_5	2.4198068e+00	x_6	2.1869661e+00
x_7	7.7291816e−01	x_8	3.7209292e−01
x_9	5.8659232e−01	x_{10}	1.7538403e+00

Many turning points are visible. The path in the full $(\lambda, x_1, \ldots, x_n)$ space is quite convoluted and gets rapidly more so as n increases. The arc length traversed as λ goes from 0 to 1 increases from about 87 for $n = 10$ to over 44,000 for $n = 130$, the largest value we tried.

6 Examples of DAETS code

In Subsection 6.1, we present a DAETS program for integrating (18). In Subsection 6.2, we show the function for evaluating the Layne Watson problem from Subsection 5.6; we omit the program for generating the numerical results in this Subsection. A comprehensive description of using the code is in the DAETS User Guide [19].

6.1 Code for integrating the pendula problem

We list and then describe the code we used for integrating (18).

```
1  #include "DAEsolver.h"
2
3  template <typename T>
4  void fcn(int n, T t, const T *x, T *f, void *p)
5  {
6    int P = *(int*)p;
7    double g = 9.8, l = 10, c = .1;
8
9    f[0] = Diff(x[0],2) + x[0]*x[2];                    //  0 = x_1'' + λ_1 x_1
10   f[1] = Diff(x[1],2) + x[1]*x[2] - g;                //  0 = y_1'' + λ_1 y_1 - g
11   f[2] =  sqr(x[0])   + sqr(x[1]) - sqr(l);           //  0 = x_1^2 + y_1^2 - L^2
12
13   for (int i = 3; i < 3*P; i+=3)
14     {
15       f[i]   = Diff(x[i],  2) + x[i]  *x[i+2];        //  0 = x_i'' + λ_i x_i
16       f[i+1] = Diff(x[i+1],2) + x[i+1]*x[i+2] - g;    //  0 = y_i'' + λ_i y_i - g
17       f[i+2] = sqr(x[i]) + sqr(x[i+1]) - sqr(l+c*x[i-1]);
18                                                       //  0 = x_i^2 + y_i^2 - (L + c λ_{i-1})^2
19     }
20 }
21
22 int main()
23 {
24   int P = 4;   // number of pendula
25   int n = 3*P; // problem size
26   DAEsolver Solver(n, DAE_FCN(fcn), &P);
27   Solver.printDAEinfo();
28   Solver.printDAEpointStructure();
29   DAEsolution x(Solver);
30   for (int j = 0; j < P; j++)
```

```
31      {
32        int d = 2*(P-j);
33        for (int i = 0; i < d; i++)
34          {
35            x.setX(3*j,   i, drand48());
36            x.setX(3*j+1,i, drand48());
37          }
38        for (int i = 0; i < d-2; i++)
39          x.setX(3*j+2,i, drand48());
40      }
41    double t0 = 0, tend = 20;
42    x.setT(t0);
43    DAEexitflag exitflag;
44    Solver.integrate(x,tend,exitflag);
45    if (exitflag!=success)
46      printDAEexitflag(exitflag);
47    else
48      {
49        cout << "\t␣t␣=␣" << x.getT() << endl;
50        for (int i = 0; i < n; i++)
51          cout << "\t␣x(" << i << ")␣=␣" << x(i,0) << endl;
52        x.printStats();
53      }
54    return 0;
55  }
```

The interface to DAETS is in the file DAEsolver.h. First, a template function for evaluating the DAE must be coded, lines 3–20. The paramers to this function are:

n : size of the problem;

t : time variable;

x : pointer to an array for storing the input variables;

f : pointer to an array for storing the residual;

p : void pointer for passing additional data to this function.

Here, the Diff operator performs the differentiation of a variable.

Then, we write a main program. First a solver is created, line 26, where we pass the size of the problem, the function fcn (DAE_FCN is a pre-defined macro), and as an optional parameter, the number of pendula. After a solver is created, we print information about the DAE and the variables and their derivatives that need to be initialized, lines 27 and 28, respectively; see also the output of this program below.

A DAEsolution object is created, line 29, and initial values are set in lines 30–40; a value for the time variable is set in line 42. The integration is performed by the call to the integrate function in line 44. If the exitflag is success, the DAEsolution object x contains solution values at tend; otherwise, x contains solution values at the reached t between t_0 and $t_{\rm end}$.

The value of the reached t is printed in line 49, and values for x_i, $i = 1, \ldots, n$, are printed in lines 50–51. Finally, statistics about the integration are printed in line 52.

The resulting output for the case of $P = 4$ pendula is

```
      DAE
        size.................12
        index ...............9
        LINEAR

Initial values must be given for:
   variable    derivatives
```

```
       0      0      1      2      3      4      5      6      7
       1      0      1      2      3      4      5      6      7
       2      0      1      2      3      4      5
       3      0      1      2      3      4      5
       4      0      1      2      3      4      5
       5      0      1      2      3
       6      0      1      2      3
       7      0      1      2      3
       8      0      1
       9      0      1
       10     0      1
       -

******************************************************************************
This program contains IPOPT, a program for large-scale nonlinear optimization.
   IPOPT is released as open source under the Common Public License (CPL).
             For more information visit www.coin-or.org/Ipopt
******************************************************************************
        t = 2.000000e+01
        x(0) = 3.370108e+00
        x(1) = 9.415008e+00
        x(2) = 9.955005e-01
        x(3) = 4.242869e+00
        x(4) = 9.165095e+00
        x(5) = 2.193503e+00
        x(6) = 1.784141e+00
        x(7) = 1.006240e+01
        x(8) = 9.899183e-01
        x(9) = 5.338172e+00
        x(10) = 8.572838e+00
        x(11) = 8.838421e-01

         CPU TIME (sec)...........0.2383
         NO STEPS.................112
            accepted..............112
            rejected..............0    * 0.00%
         STEPSIZES
            smallest..............0.15
            largest ..............0.22
         ORDER OF TAYLOR SERIES...15
         TOLERANCE
            relative..............1.0e-12
            absolute..............1.0e-12
```

6.2 Code for the Layne Watson problem

The following code describes the function in the Layne Watson problem.

```
template <typename T>
void fcn(int nplus1, T s, const T *x, T *f, void *p)
{
  int n = nplus1-1;
  T lambda = x[0];

  // Define f[0] = function S that specifies s to be arc length:
  f[0] = -1.0;
  for (int k=0; k<nplus1; k++)
    f[0] += sqr(Diff(x[k],1));
  //  0 = x_1'^2 + x_2'^2 + ... + x_n'^2

  T sum = 0.0;
  for (int k=1; k<=n; k++)
    sum += x[k];
  // sum = sum_{i=1}^n x_i

  // The algebraic equations are f[1] to f[n]
  for (int i=1; i<=n; i++)
    f[i] = x[i] - lambda * exp(cos(i*sum));   // 0 = x_i - lambda exp(cos(i sum_{i=1}^n x_i))
}
```

7 Conclusions

Comparisons and challenges.

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. It has shown itself robust in our experiments. It has as good accuracy-to-tolerance proportionality as do the codes DASSL and RADAU we compare it with, and far better on the index 3 car axis problem (Subsection 5.1). It is especially efficient at high accuracies (Subsection 5.2). Its symbolic understanding of the structure of the DAE enables it to handle high index problems (Subsection 5.3), as well as purely algebraic continuation problems (Subsection 5.6) and explicit or implicit ODEs.

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

DAETS copes well with moderately stiff problems (Subsection 5.4) because of the increasing (though bounded) stability regions of Taylor methods as the order increases.

DAETS cannot handle very large problems, very stiff problems, and problems 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* methods to DAEs: they are a sort of Taylor series from both ends of the interval at once, and for ODEs a very effective alternative to Taylor methods with the extra advantage of handling stiffness [6, 16]. It looks an interesting task in numerical analysis to devise the formulae and data structures to achieve this.

Wrong structural analysis. This event is rare in our experience, but likely to become common as SA-based methods are applied more widely. They pose another interesting and difficult task, more in computer science than in numerical analysis. In all the cases we have seen, it is possible to 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?

Future developments.

In the short term the following enhancements to DAETS are planned.

We aim to include *event location.* Namely, given a set of real event functions $g, \ldots, g_m$ of the variables and derivatives, check if any g_r has changed sign over an integration step. Then locate its zero accurately.

We aim to move to *defect-based error estimation* following the methods of Enright [8]. An approximate solution $\mathbf{u}(t)$ over the whole interval is constructed. The defect is the residual on substituting $\mathbf{u}$ into the DAE.

Although the error control of DAETS has proved robust in our tests, it is less well founded on theory than are other parts of the code, and there is evidence that it can be over-cautious. Theory indicates that defect estimation gives a more unambiguous foundation to error control for DAEs than does local error estimation.

Finally, we aim to combine this with stepsize control based on estimation of the *radius of convergence* following the algorithm of Chang and Corliss [4].

References

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

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

[5] Andreas Griewank, David Juedes, and Jean 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.

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

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

[8] P. M. Hanson and Wayne H. Enright. Controlling the defect in existing variable-order Adams codes for initial-value problems. *ACM Trans. Math. Soft.*, 9:71–97, March 1983.

[9] 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.

[10] 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`.

[11] 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.

[12] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM J. Sci. Comput.*, 14(3):677–692, 1993.

[13] 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/`.

[14] 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.

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

[16] Nedialko Stoyanov 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.

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

[18] LAPACK project. LAPACK — Linear Algebra PACKage. `www.netlib.org/lapack/`.

[19] 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.

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

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

[22] 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.

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

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

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

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