BIT
Submitted August 2005, pp. xxx–xxx

0006-3835/00/4004-0001 $15.00

SOLVING DIFFERENTIAL-ALGEBRAIC EQUATIONS BY TAYLOR SERIES (II): COMPUTING THE SYSTEM JACOBIAN*

Nedialko S. Nedialkov[1†] and John D. Pryce[2‡]

[1] *Department of Computing and Software, McMaster University, Hamilton, Ontario, L8S 4L7, Canada. email: nedialk@mcmaster.ca*

[2] *Computer Information Systems Engineering Department, Cranfield University, RMCS Shrivenham, Swindon SN6 8LA, UK. email: pryce@rmcs.cranfield.ac.uk*

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 (BIT, accepted July 2005). Numerical results have shown that this method is efficient and very accurate. Moreover, it is particularly suitable for problems that are of too high an index for present DAE solvers.

This paper develops an effective method for computing a DAE's System Jacobian, which is needed in the structural analysis of the DAE and computation of Taylor coefficients. Our method involves preprocessing of the DAE and code generation employing automatic differentiation. Theory and algorithms for preprocessing and code generation are presented.

An operator-overloading approach to computing the System Jacobian is also discussed.

AMS subject classification: 34A09, 65L80, 65L05, 41A58.

Key words: Differential-algebraic equations (DAEs), structural analysis, Taylor series, automatic differentiation.

1 Introduction.

We have developed a Taylor series (TS) method for solving numerically an initial-value problem (IVP) DAE that can be of arbitrary index, fully implicit, nonlinear, and contain derivatives of order higher than one [2].

More specifically, our method solves a DAE IVP comprising n equations $f_i = 0$ in n dependent variables $x_j = x_j(t)$, with t a scalar independent variable. We write informally

$$f_i(\, t, \text{ the } x_j \text{ and derivatives of them}\,) = 0, \quad 1 \le i \le n. \tag{1.1}$$

*Received .... Revised .... Communicated by ....

†This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

‡This work was supported in part by grants from the Leverhulme Trust and the Engineering and Physical Sciences Research Council of the UK.

The f_i are assumed sufficiently smooth. They can be arbitrary expressions built from the x_j and t using $+, -, \times, \div$, other standard functions, and the differentiation operator d^p/dt^p.

To solve (1.1), we use automatic differentiation (AD) to generate functions for evaluating the Taylor coefficients (TCs) of the equations f_i, and by equating these coefficients to zero, we solve implicitly for the TCs of the solution components $x_j(t)$. Then we sum these coefficients with appropriate stepsize to find an approximate Taylor series solution, which we project to satisfy the constraints of the DAE. We repeat this process on each integration step in a standard time-stepping manner.

To determine what equation to solve, and for which TCs of the solution, we apply Pryce's structural analysis (SA) [4]: we preprocess the given DAE to find the *signature matrix* and then the *offsets* of the problem. These offsets prescribe the overall process for computing TCs, as well as how to form the *System Jacobian* $\mathbf{J}$ that is central to both the theory and the algorithm.

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. Our method does not find high index inherently hard. The SA derives a *structural index*, which is the same as that found by the well-known method of Pantelides [3], and is always $\geq \nu_d$.

Underlying theory, algorithms, numerical results, and implementation issues are presented in [2]. This method is implemented in the authors' DAETS code, which is written in standard `C++`. The numerical results in [2] show DAETS can be very accurate, efficient, and particularly suitable for problems that are of too high an index for existing methods and solvers.

This is the second paper on the theory behind DAETS. We focus on preprocessing a DAE and computing its System Jacobian. Our main contribution is the theory and algorithms of a source-code translation method for evaluating this Jacobian. We also describe a method for computing it based on operator overloading. Algorithmic details for the preprocessing phase of constructing a signature matrix of a DAE are given.

Section 2 summarizes Pryce's SA. Section 3 illustrates how TCs are calculated. The computation of the signature matrix is presented in Section 4. Section 5 develops an efficient method, based on source-code translation, for evaluating $\mathbf{J}$. Section 6 describes an operator-overloading approach for computing $\mathbf{J}$. Concluding remarks are in Section 7.

2 Summary of Pryce's SA.

When computing TCs for the solution of a DAE, we employ Pryce's SA to determine what equations to solve and for which coefficients of the solution. The steps of this SA are outlined below.

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

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

2. Find a *highest-value transversal* (HVT) in Σ. A *transversal* of an $n \times n$ matrix is a set of n positions in this matrix with one entry in each row and each column. A HVT is a transversal T that makes $\sum_{(i,j)\in T} \sigma_{ij}$ as large as possible.

3. Calculate the *offsets* of the problem. These are two nonnegative integer n-vectors $\mathbf{c}$ and $\mathbf{d}$ that satisfy

$$d_j - c_i = \sigma_{ij} \quad \text{for all } (i,j) \text{ in some HVT } T \quad \text{and} \tag{2.1}$$
$$d_j - c_i \geq \sigma_{ij} \quad \text{for all } i,j = 1,\ldots,n. \tag{2.2}$$

   The conditions (2.1, 2.2) are independent of the T chosen. However, the offsets are never unique. When computing TCs, it is advantageous to choose the *smallest* or *canonical* offsets [2], smallest being in the sense of $\mathbf{a} \leq \mathbf{b}$ if $a_i \leq b_i$ for each i.

4. Form the *System Jacobian* of (1.1):

$$\mathbf{J} = \frac{\partial\left(f_1^{(c_1)},\ldots,f_n^{(c_n)}\right)}{\partial\left(x_1^{(d_1)},\ldots,x_n^{(d_n)}\right)}. \tag{2.3}$$

   By results in [4], (2.3) has the equivalent reformulation:

$$\mathbf{J}_{ij} \;=\; \frac{\partial f_i}{\partial x_j^{(d_j-c_i)}} \;=\; \begin{cases} \dfrac{\partial f_i}{\partial x_j^{(\sigma_{ij})}} & \text{if } d_j - c_i = \sigma_{ij} \quad \text{and} \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

Informally, a consistent initial point is the values of an appropriate set of the x_j and their derivatives, at a time t^*, that specify a unique solution; see [2] for a more rigorous discussion. If $\mathbf{J}$ is nonsingular at a consistent point, then the SA has succeeded, and the DAE is solvable in a neighborhood of this point.

The subject of this paper is steps 1 and 4. Steps 2 and 3 comprise a *linear assignment problem* (LAP), solvable by a suitable LAP code; see [2, 4].

EXAMPLE 2.1. The simple pendulum is a DAE of differentiation-index 3:

$$\begin{aligned} 0 &= f = x'' + x\lambda \\ 0 &= g = y'' + y\lambda - G \\ 0 &= h = x^2 + y^2 - L^2. \end{aligned} \tag{2.5}$$

Here gravity $G>0$ and length $L>0$ of pendulum are constants, and the dependent variables are the coordinates $x(t)$, $y(t)$ and the Lagrange multiplier $\lambda(t)$.

For (2.5), there are two HVTs, marked $^\bullet$ and $^\circ$ in the tableau below. The canonical offsets are $\mathbf{c} = (0, 0, 2)$ and $\mathbf{d} = (2, 2, 0)$:

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

The system Jacobian is

$$\mathbf{J} = \begin{bmatrix} \partial f/\partial x'' & 0 & \partial f/\partial \lambda \\ 0 & \partial g/\partial y'' & \partial g/\partial \lambda \\ \partial h/\partial x & \partial h/\partial y & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 2x & 2y & 0 \end{bmatrix}. \tag{2.6}$$

Now $\mathbf{J}$ is nonsingular since its determinant is $-2(x^2 + y^2) = -2L^2 \neq 0$.

3 Computing TCs.

The offsets c_i and d_j prescribe how to organize the computation of TCs for the x_j. Denote the rth TC of x_j by $(x_j)_r$ and the rth TC of f_i by $(f_i)_r$.

We compute TCs in stages starting from stage $k_d = -\max_j d_j$. At each stage $k = k_d, k_d + 1, \ldots$, we solve a system of equations

$$(f_i)_{k+c_i} = 0 \quad \text{for all } i \text{ such that} \quad k + c_i \geq 0 \tag{3.1}$$

to determine values for

$$(x_j)_{k+d_j} \quad \text{for all } j \text{ such that} \quad k + d_j \geq 0. \tag{3.2}$$

All previously computed $(x_j)_r$ in any equation (3.1) are *treated as constants*.

The equations in (3.1) are generated using AD for computing TCs of explicit functions, here f_i. For example, if sufficient TCs of x_j and x_k are known, we can evaluate the pth TCs for $x_j + x_k$, $x_j \cdot x_k$, and $x_j^{(d)}$ by

$$(x_j + x_k)_p = (x_j)_p + (x_k)_p, \tag{3.3}$$

$$(x_j \cdot x_k)_p = \sum_{r=0}^{p} (x_j)_r (x_k)_{p-r}, \quad \text{and} \tag{3.4}$$

$$\left(x_j^{(d)}\right)_p = (p+1)(p+2)\cdots(p+d) \cdot (x_j)_{p+d}. \tag{3.5}$$

Similar formulas can be written for division and the standard functions.

Then, the values in (3.2) are found by an implicit solution process. The Jacobian that we have to form at stage k has entries

$$\frac{\partial (f_i)_{k+c_i}}{\partial (x_j)_{k+d_j}} = \frac{\partial f_i^{(k+c_i)}/(k+c_i)!}{\partial x_j^{(k+d_j)}/(k+d_j)!} = \frac{(k+d_j)!}{(k+c_i)!}\,\frac{\partial f_i^{(k+c_i)}}{\partial x_j^{(k+d_j)}},$$

where $k + c_i \geq 0$ and $k + d_j \geq 0$.

Denote by $\mathbf{J}_k$ the Jacobian formed from $\partial f_i^{(k+c_i)}/\partial x_j^{(k+d_j)}$, for those i and j for which $k + c_i \geq 0$ and $k + d_j \geq 0$. We show in Subsection 5.1 that for $k < 0$, $\mathbf{J}_k$ is the submatrix of $\mathbf{J}$ formed by deleting rows i where $k+c_i < 0$ and columns j where $k + d_j < 0$. If $k \geq 0$ then $\mathbf{J}_k = \mathbf{J}$, since no rows or columns are deleted.

EXAMPLE 3.1. For the pendulum example (2.5), we write TCs without parentheses for brevity: x_r rather than $(x)_r$, etc. Then using (3.1, 3.2) and (3.3–3.5) applied to the pendulum equations, we obtain the scheme:

stage	uses equations	to obtain
$k = -2$	$0 = h_0 = x_0^2 + y_0^2 - L^2$	x_0, y_0
$k = -1$	$0 = h_1 = 2x_0x_1 + 2y_0y_1$	x_1, y_1
$k = 0$	$0 = f_0 = 2x_2 + x_0\lambda_0$ $0 = g_0 = 2y_2 + y_0\lambda_0 - G$ $0 = h_2 = 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2$	x_2, y_2, λ_0
$\vdots$	$\vdots$	$\vdots$
$k = p$	$0 = f_p = \ldots$ $0 = g_p = \ldots$ $0 = h_{p+2} = \ldots$	$x_{p+2}, y_{p+2}, \lambda_p$

At stages $k = -2$ and $k = -1$,

$$\frac{\partial h_0}{\partial (x_0, y_0)} = \frac{\partial h_1}{\partial (x_1, y_1)} = [2x_0, 2y_0] = \mathbf{J}_{-2} = \mathbf{J}_{-1}.$$

At stage $k = 0$, we have the linear system

$$0 = \begin{bmatrix} 2 & 0 & x_0 \\ 0 & 2 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix} \begin{pmatrix} x_2 \\ y_2 \\ \lambda_0 \end{pmatrix} + \begin{pmatrix} 0 \\ -G \\ x_1^2 + y_1^2 \end{pmatrix} = \mathbf{A}_0\xi + \mathbf{b}, \text{ say.}$$

Matrix $\mathbf{A}_0$ is a diagonally-scaled version of the pendulum's System Jacobian $\mathbf{J}$, given in (2.6). Namely

$$\mathbf{A}_0 = \frac{\partial (f_0, g_0, h_2)}{\partial (x_2, y_2, \lambda_0)} = \text{diag}[1, 1, 2]^{-1}\,\mathbf{J}\,\text{diag}[2, 2, 1].$$

Similarly for $k > 0$, we solve a linear system $f_k, g_k, h_{k+2} = 0$ to determine $x_{k+2}, y_{k+2}, \lambda_k$. The matrix in each of these systems is a diagonally-scaled $\mathbf{J}$.

Generally, (3.1) can be nonlinear for $k \leq 0$ and underdetermined for $k < 0$. For $k > 1$, it is always linear with a diagonally-scaled $\mathbf{J}$. For details, see [2].

4 Computing Σ.

First we define a signature vector of a variable in the code list of a DAE. Then, we give a Lemma relating the signature vectors of these variables. Finally, we outline an algorithm for computing Σ based on this Lemma.

DEFINITION 4.1 (SIGNATURE VECTOR). *Let v be a variable in the code list. The* signature vector *of v is the n vector $\sigma(v)$ with j^{th} component*

$$\sigma_j(v) = \begin{cases} -\infty \text{ if } v \text{ does not depend on } x_j\text{; or} \\ \text{the highest order of derivative of } x_j \text{ on which } v \text{ formally depends.} \end{cases}$$

By "formally" we mean dependence in the code list without simplifications. For instance, the order of highest derivative of x in $x' + x - x'$ is 1, not 0; similarly, the order of the highest derivative of x in $(xy)' - x'y$ is 1, not 0.

LEMMA 4.1. *Let v be a variable in the code list.*

(i) If v is an input-variable x_j then

$$\sigma_l(v) = \sigma_l(x_j) = \begin{cases} 0 & \text{if } l = j \\ -\infty & \text{if } l \neq j. \end{cases} \tag{4.1}$$

(ii) If v is a constant then

$$\sigma_l(v) = -\infty \quad \text{for } l = 1, \ldots, n.$$

(iii) If v is an algebraic function of a set U of variables u, then

$$\sigma_l(v) = \max_{u \in U} \sigma_l(u) \quad \text{for } l = 1, \ldots, n. \tag{4.2}$$

(iv) If $v \equiv d^p u/dt^p$ then

$$\sigma_l(v) = \sigma_l(u) + p \quad \text{for } l = 1, \ldots, n. \tag{4.3}$$

The proof is trivial, and we omit it. This Lemma immediately gives:

ALGORITHM 4.1 (SIGNATURE MATRIX).

Input
code list encoding a DAE

Output
signature matrix Σ

Compute
for each term v in the code list
  if v is an input-variable x_j then
    $\sigma_l(v) \leftarrow 0$ if $l = j$
    $\sigma_l(v) \leftarrow -\infty$ for $l \neq j$
  elseif v is a constant then
    $\sigma_l(v) \leftarrow -\infty$ for $l = 1, \ldots, n$

```
    elseif v is an algebraic function of a set U of previous variables u then
        σ_l(v) ← max_{u∈U} σ_l(u)  for  l = 1, ..., n
    elseif v ≡ d^p u/dt^p then
        σ_l(v) ← σ_l(u) + p  for  l = 1, ..., n
for i = 1, ..., n
    (ith row of Σ) ← σ(f_i)
```

By the remark after Definition 4.1, Algorithm 4.1 may overestimate the "true" Σ, but never underestimates it. A detailed study, showing that such overestimations do not deceive the numerical method, is given in [2].

The DAETS solver implements this algorithm through operator overloading.

5 Computing Jacobians through source-code translation.

We show how to compute efficiently $\mathbf{J}_k$ for $k \leq 0$ from the code list of a DAE. Our method is suitable for an implementation based on source-text translation. We call it a *single-pass* method.

In the next subsection, we start with a result quoted as Lemma 3.7 in [4], but used by Griewank in [1], and establish the relation between $\mathbf{J}_k$ and $\mathbf{J}$. In subsection 5.2, we define the offset of a variable in the code list and give an algorithm for computing such offsets. Subsection 5.3 states a Theorem on which our single-pass method is based, shows an algorithm implementing it, and illustrates this method on an example.

5.1 Griewank's Lemma

LEMMA 5.1 (GRIEWANK). *Let v be a function of the $x_j(t)$ and their derivatives $(j = 1, \ldots, n)$. Denote $v^{(p)} = d^p v/dt^p$. If v does not depend on any derivative of x_j higher than the q^{th}, then*

$$\frac{\partial v'}{\partial x_j^{(q+1)}} = \frac{\partial v}{\partial x_j^{(q)}}. \tag{5.1}$$

Hence by iterating

$$\frac{\partial v^{(p)}}{\partial x_j^{(q+p)}} = \frac{\partial v}{\partial x_j^{(q)}} \quad \textit{for all} \quad p \geq 0. \tag{5.2}$$

EXAMPLE 5.1. Consider $v = xx' + yy' = v(x, x', y, y')$, where the dependent variables are x, y. Then $v' = xx'' + x'^2 + yy'' + y'^2 = v'(x, x', x'', y, y', y'')$. Griewank's Lemma asserts that

$$\frac{\partial v'}{\partial x''} = \frac{\partial v}{\partial x'} \quad \text{and} \quad \frac{\partial v'}{\partial x'''} = \frac{\partial v}{\partial x''}$$

(both sides are zero), but

$$\frac{\partial v'}{\partial x'} \neq \frac{\partial v}{\partial x},$$

since a derivative of x higher than x itself occurs in v.

By the definition of σ_{ij}, and because $d_j - c_i \geq \sigma_{ij}$, f_i does not depend on any derivative of x_j higher than $(d_j - c_i)$. Applying (5.2), when $k + c_i \geq 0$ and $k + d_j \geq 0$,

$$\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j^{(d_j - c_i)}} = \frac{\partial f_i^{(k+c_i)}}{\partial x_j^{(k+d_j)}} = (\mathbf{J}_k)_{ij}.$$

This gives

LEMMA 5.2. *For $k < 0$, $\mathbf{J}_k$ is the submatrix of $\mathbf{J}$ comprising the rows for which $k + c_i \geq 0$ and columns for which $k + d_j \geq 0$. For all $k \geq 0$, $\mathbf{J}_k = \mathbf{J}$.*

For example, for the pendulum,

$$\mathbf{J}_1 = \frac{\partial(f', g', h''')}{\partial(x''', y''', \lambda')} = \frac{\partial(f, g, h'')}{\partial(x'', y'', \lambda)} = \mathbf{J}.$$

5.2 Code offsets.

Denote

$$\nabla_k = \left(\frac{\partial}{\partial x_1^{(k+d_1)}}, \ldots, \frac{\partial}{\partial x_n^{(k+d_n)}} \right). \tag{5.3}$$

If $k+d_j < 0$, the jth component of ∇_k is taken as 0. For $k+c_i \geq 0$, $\nabla_k \left(f_i^{(k+c_i)} \right)$ is the ith row of $\mathbf{J}$.

The following definitions will be convenient later. Below v denotes a variable in the code list of a DAE.

DEFINITION 5.1. *We say that ∇_k* covers *v if $k+d_j \geq \sigma_j(v)$ for all $j = 1, \ldots, n$. We say that ∇_k* touches *v if*

- *∇_k covers v, and*
- *$k + d_j = \sigma_j(v)$ for at least one j.*

The offset *of v is the unique value $\alpha(v) \geq 0$ such that $\nabla_{-\alpha(v)}$ touches v. Equivalently,*

$$\alpha(v) = \min_j \big(d_j - \sigma_j(v)\big) \geq 0. \tag{5.4}$$

The computation of these offsets is based on

LEMMA 5.3.

(i) If $v \equiv x_j$, then

$$\alpha(v) = \alpha(x_j) = d_j.$$

(ii) If v is an output variable f_i, then

$$\alpha(v) = \alpha(f_i) = c_i.$$

(iii) If v is an algebraic function of a set U of variables u, then

$$\alpha(v) = \min_{u \in U} \alpha(u).$$

(iv) If $v \equiv d^p u/dt^p$, then

$$\alpha(v) = \alpha(u) - p.$$

PROOF. (i) follows from (4.1) and (5.4).

For (ii), consider the defining relation (2.2) that $d_j - c_i \geq \sigma_{ij}$ for all i, j, with equality on a transversal. Since $\sigma_{ij} = \sigma_j(f_i)$, for fixed i, one has $d_j - \sigma_{ij} = d_j - \sigma_j(f_i) \geq c_i$ for all j. By the transversality condition $d_j - c_i = \sigma_{ij}$, at least one j gives $d_j - \sigma_j(f_i) = c_i$, and therefore $\alpha(f_i) = c_i$.

For (iii), using (5.4) twice and (4.2) once, we obtain

$$\begin{aligned}
\alpha(v) &= -\max_j \big(\sigma_j(v) - d_j\big) \\
&= -\max_j \Big(\big(\max_{u \in U} \sigma_j(u)\big) - d_j\Big) \\
&= -\max_j \Big(\max_{u \in U} \big(\sigma_j(u) - d_j\big)\Big) \\
&= \min_{u \in U} \Big(-\max_j \big(\sigma_j(u) - d_j\big)\Big) \\
&= \min_{u \in U} \alpha(u).
\end{aligned}$$

Finally, (iv) comes from (4.3) and (5.4). □

This Lemma justifies:

ALGORITHM 5.1 (CODE OFFSETS).

Input

code list of a DAE

offset vectors **c** and **d**

Output

$\alpha(v)$ for each variable v in the code list

Compute

```
for each variable v in the code list
    if v ≡ x_j then
        α(v) ← d_j
    elseif v is an algebraic function of a set U of variables u then
        α(v) ← min_{u∈U} α(u)
    elseif v ≡ d^p u/dt^p then
        α(v) ← α(u) − p
```

At termination of this algorithm, by Lemma 5.3, $\alpha(f_i) = c_i$.

5.3 Single-pass method.

Now suppose that ∇_k covers v. This is equivalent to

$$k + d_j \geq \sigma_j(v) \quad \text{for each} \quad j,$$

equivalently

$$k \geq -\alpha(v),$$

by Definition 5.1. Setting $q = k + d_j$ in (5.2) of Griewank's Lemma gives

$$\frac{\partial v^{(p)}}{\partial x_j^{(k+d_j+p)}} = \frac{\partial v}{\partial x_j^{(k+d_j)}} \quad \text{for each } j. \tag{5.5}$$

Using (5.5) in the definition (5.3) of ∇_k, we have

$$\nabla_{k+p}\left(\frac{d^p v}{dt^p}\right) = \nabla_k(v) \quad \text{if } \; k = -\alpha(v), \text{ that is } \nabla_k \text{ touches } v; \tag{5.6}$$

while

$$\nabla_k(v) = 0 \quad \text{if } \; k > -\alpha(v), \text{ that is } \nabla_k \text{ covers but does not touch } v, \tag{5.7}$$

since then all differentiations are with respect to variables v does not depend on.

In addition, each ∇_k obeys the normal rules of a gradient operation: for $v = x + y$ and $v = xy$,

$$\nabla_k(x+y) = \nabla_k(x) + \nabla_k(y) \quad \text{and} \quad \nabla_k(xy) = \nabla_k(x)y + x\nabla_k(y), \tag{5.8}$$

respectively. More generally, if v is an algebraic function $e(x, y, \ldots)$ of code list variables then

$$\nabla_k(v) = \frac{\partial e}{\partial x}\nabla_k(x) + \frac{\partial e}{\partial y}\nabla_k(y) + \cdots . \tag{5.9}$$

The factors $\partial e/\partial x, \ldots$ are to be evaluated at $(x, y, \ldots)$ and thus are also functions of the input variables and their derivatives. If ∇_k *touches* v, then by (4.2) it must *cover* each of $x, y, \ldots$. From (5.7), any term in (5.9) for which ∇_k does not touch the corresponding variable is zero.

The following Theorem, most of which has been proved already, summarizes the facts on which the single-pass method relies.

THEOREM 5.4.

(i) If $v \equiv x_j$, then

$$\nabla_{-\alpha(v)}(v) = \nabla_{-d_j}(x_j) = e_j,$$

the j^{th} unit row vector.

(ii) If v is an output variable f_i, then

$$(i^{th} \text{ row of } \mathbf{J}) = \nabla_{-c_i}(f_i).$$

(iii) If v is an algebraic function $e(x, y, \ldots)$ of a set U of previously computed code list variables, then

$$\nabla_{-\alpha(v)}(v) = \sum_{\substack{u \in U \\ \alpha(u) = \alpha(v)}} \frac{\partial e}{\partial u}\, \nabla_{-\alpha(u)}(u).$$

(iv) If $v \equiv d^p u/dt^p$, *then*

$$\nabla_{-\alpha(v)}(v) = \nabla_{-\alpha(u)}(u).$$

PROOF. (i) When $v = x_j$, $\alpha(v) = d_j$ by Lemma 5.3(i). From (5.3), the jth component of $\nabla_{-\alpha(v)}(v) = \nabla_{-d_j}(x_j)$ is

$$\frac{\partial x_j}{\partial x_j^{(k+d_j)}} = \frac{\partial x_j}{\partial x_j^{(d_j-d_j)}} = \frac{\partial x_j}{\partial x_j} = 1.$$

The other components are clearly 0.

(ii) By Lemma 5.3(ii), $\alpha(f_i) = c_i$; hence $\nabla_{-\alpha(v)}(v) = \nabla_{-c_i}(f_i)$. From (5.3, 5.6),

$$\nabla_{-c_i}(f_i) = \nabla_0\left(f_i^{(c_i)}\right) = \left(\frac{\partial f_i^{(c_i)}}{\partial x_1^{(d_1)}}, \ldots, \frac{\partial f_i^{(c_i)}}{\partial x_n^{(d_n)}}\right),$$

which is the definition of the ith row of the system Jacobian; cf. (2.3).

(iii) re-states what was stated and justified in the paragraph containing (5.9), while (iv) comes from (5.6). □

The resulting method is shown in Algorithm 5.2. We denote $\nabla_{-\alpha(v)}(v)$ by $G(v)$. In practice, the values of v and of $G(v)$ should be computed simultaneously in the same loop. For clarity, this simultaneous computation is not shown here.

ALGORITHM 5.2 (SYSTEM JACOBIAN).

Input

code list of a DAE

Output

code list for computing $\mathbf{J}$

Compute

Σ by Algorithm 4.1
$\mathbf{c}$, $\mathbf{d}$ by solving an LAP
$\alpha(v)$ for each variable v in the code list by Algorithm 5.1

for each variable v in the code list
  if $v \equiv x_i$ then
    $G(v) \leftarrow e_i$
  elseif v is an algebraic function $e(x, y, \ldots)$ of a set U of variables u then

$$G(v) \leftarrow \sum_{\substack{u \in U \\ \alpha(u)=\alpha(v)}} \frac{\partial e}{\partial u}\, G(u)$$

  elseif $v \equiv d^p u/dt^p$ then
    $G(v) \leftarrow G(u)$

for $i = 1, \ldots, n$
  (ith row of $\mathbf{J}$) $\leftarrow G(f_i)$

EXAMPLE 5.2. Consider the pendulum problem, but with the first equation changed to

$$f = \left[(x^2)'x\right]' + (x^2)' + x^2\lambda = 0.$$

Code list	$\sigma(v)$	$\alpha(v)$	Calculating $\nabla_{-\alpha(v)}(v)$	Gradient code list
x = input	0 – –	2	$\nabla_{-2}(x) = (1,0,0)$	$X = \text{unit}(1)$
y = input	– 0 –	2	$\nabla_{-2}(y) = (0,1,0)$	$Y = \text{unit}(2)$
λ = input	– – 0	0	$\nabla_0(\lambda) = (0,0,1)$	$\Lambda = \text{unit}(3)$
$v_1 = x^2$	0 – –	2	$\nabla_{-2}(v_1) = 2x\nabla_{-2}(x)$ $= (2x,0,0)$	$V_1 = 2xX$
$v_2 = v_1\lambda$	0 – 0	0	$\nabla_0(v_2) = \cancel{\nabla_0(v_1)\lambda} + v_1\nabla_0(\lambda)$ $= (0,0,x^2)$	$V_2 = v_1\Lambda$
$v_3 = v_1'$	1 – –	1	$\nabla_{-1}(v_3)$ same as $\nabla_{-2}(v_1)$	
$v_4 = v_3x$	1 – –	1	$\nabla_{-1}(v_4) = \nabla_{-1}(v_3)x + \cancel{v_3\nabla_{-1}(x)}$ $= (2x^2,0,0)$	$V_4 = xV_1$
$v_5 = v_4'$	2 – –	0	$\nabla_0(v_5)$ same as $\nabla_{-1}(v_4)$	
$v_6 = v_5 + v_3$	2 – 0	0	$\nabla_0(v_6) = \nabla_0(v_5) + \cancel{\nabla_0(v_3)}$ $= (2x^2,0,0)$	$V_6 = V_4$
$f_1 = v_6 + v_2$	2 – 0	0	$\nabla_0(f_1) = \nabla_0(v_6) + \nabla_0(v_2)$ $= (2x^2,0,x^2)$	$F_1 = V_6{+}V_2$
$v_7 = y''$	– 2 –	0	$\nabla_0(v_7)$ same as $\nabla_{-2}(y)$	
$v_8 = y\lambda$	– 0 0	0	$\nabla_0(v_8) = \cancel{\nabla_0(y)\lambda} + y\nabla_0(\lambda)$ $= (0,0,y)$	$V_8 = y\Lambda$
$f_2 = v_7 + v_8 - G$	– 2 0	0	$\nabla_0(f_2) = \nabla_0(v_7) + \nabla_0(v_8)$ $= (0,1,y)$	$F_2 = Y{+}V_8$
$v_9 = y^2$	– 0 –	2	$\nabla_{-2}(v_9) = 2y\nabla_{-2}(y)$ $= (0,2y,0)$	$V_9 = 2yY$
$f_3 = v_1 + v_9 - L^2$	0 0 –	2	$\nabla_{-2}(f_3) = \nabla_{-2}(v_1) + \nabla_{-2}(v_9)$ $= (2x,2y,0)$	$F_3 = V_1{+}V_9$

Figure 5.1: Computing f_1, f_2 and f_3 and their gradients. Sample code to compute Jacobian is given in the last column. The terms that are zero because of (5.7) are crossed out. Function unit(i) returns the ith unit row vector.

This change does not alter the signature matrix, so the offsets remain unchanged: $\mathbf{c} = (0,0,2)$ and $\mathbf{d} = (2,2,0)$.

Figure 5.1 illustrates how gradient code can be generated for a code list that computes $f_1 \equiv f$, $f_2 \equiv g = y'' + y\lambda - G$, and $f_3 \equiv h = x^2 + y^2 - L^2$. The last column of the table shows how the code list in the first column can be augmented, line by line, to compute the rows of $\mathbf{J}$. Here, gradients are denoted by uppercase letters. All terms that are known to be zero are omitted in the gradient code.

Where the phrase "X same as Y" occurs in the penultimate column (from a use of (5.6)), this is taken to mean that no copy is made: X is just another name for Y. Otherwise, no code optimization has been done.

The mathematical relations between the $\nabla_r(v)$, and the sample code above,

Code list	$\alpha(v)$	Gradient code list
$x = \text{input}$	2	$X = \text{unit}(1)$
$y = \text{input}$	2	$Y = \text{unit}(2)$
$v_1 = x^2$	2	$V_1 = 2xX$
$v_9 = y^2$	2	$V_9 = 2yY$
$f_3 = v_1 + v_9 - L^2$	2	$F_3 = V_1 + V_9$
		if $k < 0$, exit
$v_3 = v_1'$	1	
$v_4 = v_3 x$	1	$V_4 = xV_1$
$\lambda = \text{input}$	0	$\Lambda = \text{unit}(3)$
$v_2 = v_1 \lambda$	0	$V_2 = v_1 \Lambda$
$v_5 = v_4'$	0	
$v_6 = v_5 + v_3$	0	$V_6 = V_4$
$f_1 = v_6 + v_2$	0	$F_1 = V_6 + V_2$
$v_7 = y''$	0	
$v_8 = y\lambda$	0	$V_8 = y\Lambda$
$f_2 = v_7 + v_8 - G$	0	$F_2 = Y + V_8$

Figure 5.2: The code from Figure 5.1 reordered for computation of $\mathbf{J}_k$ when $k < 0$. For clarity, columns 2 and 4 of the table in Figure 5.1 are not shown.

assume that all vectors involved have length n. That is, columns with $k + d_j < 0$ are not suppressed, but contain padding zeros. This is for notational convenience — a practical implementation could use a sparse storage scheme.

The case $k < 0$. When $k < 0$, stage k of the overall Taylor coefficient generation algorithm [2] requires the partial Jacobian $\mathbf{J}_k$, which comprises the rows i and columns j of $\mathbf{J}$ for which $k + c_i \geq 0$ and $k + d_j \geq 0$ (Lemma 5.2). In this case, we wish to execute only the gradient code list corresponding to $\mathbf{J}_k$, not the code list for the whole $\mathbf{J}$.

To let Algorithm 5.2 handle efficiently the case $k < 0$, just reorder the code list such that the lines that compute the f_i are in descending order of c_i. In doing so, one must ensure that no variable comes before one that it depends on. A simple way to do so is to perform a stable descending sort on the code list with $\alpha(v)$ as key (a sorting method is stable when items with the same value of the sort key appear in the output in the same order as they do in the input).

Then if $k < 0$, execute only the initial segment of the code list up to where the needed f_i are computed. To achieve this, it is *sufficient* to execute just those lines for which $k + \alpha(v) \geq 0$.

EXAMPLE 5.3. Figure 5.2 shows the sorted code list from Figure 5.1. The full System Jacobian is

$$\mathbf{J} = \begin{bmatrix} \nabla_0(f_1) \\ \nabla_0(f_2) \\ \nabla_{-2}(f_3) \end{bmatrix} = \begin{bmatrix} 2x^2 & 0 & x^2 \\ 0 & 1 & y \\ 2x & 2y & 0 \end{bmatrix}.$$

Jacobian $\mathbf{J}_{-2}$ comprises the x, y columns (for which $-2 + d_j \geq 0$) and the f_3 row (for which $-2 + c_i \geq 0$): $\mathbf{J}_{-2} = [\,2x\ 2y\,]$. This Jacobian is calculated from the first block of rows in Figure 5.2. Jacobian $\mathbf{J}_{-1}$ happens to be the same.

Note that executing the lines for which $k+\alpha(v) \geq 0$ is not always optimal. For instance, when $k = -1$, this would cause the lines with $\alpha(v) = 1$ in Figure 5.2 to be needlessly executed.

Finally, the last example makes clear that, using source-text translation, this method can produce very concise code for computing the matrices $\mathbf{J}_k$, with at most one vector operation inserted per scalar elementary operation in the code list of the DAE.

6 Computing Jacobians through operator overloading.

In DAETS, the TCs in (3.1) are evaluated using operator overloading. Simultaneously with their computation, the necessary Jacobians are evaluated, again through operator overloading. This approach is carried out using the `C++` package FADBAD++ [5]. It provides a template class `F` implementing the forward mode of AD. If `F` is instantiated with `double`, the class `F<double>` implements the forward mode on `C/C++ double`'s by operator overloading. FADBAD++ also provides a template class `T` implementing a TC computation. This class can be instantiated with a scalar data type, for example `double` as `T<double>`. Furthermore, it can be instantiated with a "differentiation" type `F<double>`, resulting in a `T< F<double> >` class. When a TC computation with `T< F<double> >` objects is performed, it computes both TCs and their gradients, that is the $\nabla_k(v)$ of the last section.

Example 6.1. For illustration, consider again the pendulum (2.5) with the first equation changed to

$$f = \left[(x^2)'x\right]' + (x^2)' + x^2\lambda = 0. \tag{6.1}$$

Denote

$$\nabla = \left(\frac{\partial}{\partial x_2}, \frac{\partial}{\partial y_2}, \frac{\partial}{\partial \lambda_0}\right).$$

We show how $\nabla f_0, \nabla g_0$, and ∇h_2 are evaluated.

Applying (3.3, 3.4, 3.5) to (6.1), the computation of the expression for the zero-order TC of f using operator overloading would be

$$\begin{aligned}
f_0 &= \Big(\left[(x^2)'x\right]' + (x^2)' + x^2\lambda\Big)_0 \\
&= \Big(\left[(x^2)'x\right]'\Big)_0 + \big((x^2)'\big)_0 + (x^2\lambda)_0 \\
&= \big((x^2)'x\big)_1 + (x^2)_1 + (x^2)_0\lambda_0 \\
&= \big((x^2)'\big)_1 x_0 + \big((x^2)'\big)_0 x_1 + 2x_0x_1 + x_0^2\lambda_0 \\
&= 2(x^2)_2\, x_0 + (x^2)_1 x_1 + 2x_0x_1 + x_0^2\lambda_0 \\
&= 2(2x_0x_2 + x_1^2)x_0 + (2x_0x_1)x_1 + 2x_0x_1 + x_0^2\lambda_0.
\end{aligned} \tag{6.2}$$

Now, each TC x_0, x_1, x_2, and λ_0 is represented by an `F<double>` object, which contains the corresponding TC value and a gradient with respect to (x_2, y_2, λ_0). On input, we associate with x_2, y_2, λ_0 gradients $\nabla x_2 = (1,0,0)$, $\nabla y_2 = (0,1,0)$, and $\nabla\lambda_0 = (0,0,1)$, respectively. For the rest of the TCs, we associate gradient vectors with zero components.[1]

The arithmetic operations between `F<double>`'s follow the rules of gradient computation (5.8). Then, with the evaluation of (6.2), the following computation involving nonzero gradients occurs:

$$\begin{aligned}
&\text{line 4 in (6.2):} && x_0^2\,\nabla\lambda_0 = x_0^2\cdot(0,0,1) = (0,0,x_0^2)\\
&\text{line 6 in (6.2):} && 2(2x_0\nabla x_2)x_0 + x_0^2\,\nabla\lambda_0 = 2\big(2x_0\cdot(1,0,0)\big)x_0 + (0,0,x_0^2)\\
& && \quad = (4x_0^2, 0, x_0^2) = \nabla f_0.
\end{aligned}$$

This may seem simpler than the source-code translation approach, but generally it is less efficient. For example, consider the term $2x_0x_1$ in the fourth line of (6.2). The multiplication of the objects corresponding to x_0 and x_1 results in the calculation

$$\nabla(x_0x_1) = x_0\nabla x_1 + \nabla x_0\, x_1 = x_0\cdot(0,0,0) + x_1\cdot(0,0,0) = (0,0,0),$$

since x_0 and x_1 are represented in `F<double>` objects, and their multiplication involves gradient operations.

Similarly, we have for the g function

$$g_0 = (y'' + y\lambda - G)_0 = 2y_2 + y_0\lambda_0.$$

The operations between the `F<double>`'s corresponding to y_0, y_2, and λ_0 generate the calculation

$$\begin{aligned}
\nabla g_0 &= 2\nabla y_2 + \nabla y_0\lambda_0 + y_0\nabla\lambda_0\\
&= 2(0,1,0) + \lambda_0\cdot(0,0,0) + y_0\cdot(0,0,1)\\
&= (0,2,y_0).
\end{aligned}$$

Finally, we compute for h,

$$\begin{aligned}
h_2 &= \big(x^2 + y^2 - L\big)_2 = \big(x^2\big)_2 + \big(y^2\big)_2\\
&= 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2
\end{aligned}$$

and ($(0,0,0)$ terms are omitted)

$$\begin{aligned}
\nabla h_2 &= 2x_0\nabla x_2 + 2y_0\nabla y_2\\
&= 2x_0\cdot(1,0,0) + 2y_0\cdot(0,1,0)\\
&= (2x_0, 2y_0, 0).
\end{aligned}$$

Hence

$$\frac{\partial(f_0,g_0,h_2)}{\partial(x_2,y_2,\lambda_0)} = \begin{bmatrix} 4x_0^2 & 0 & x_0^2\\ 0 & 2 & y_0\\ 2x_0 & 2y_0 & 0\end{bmatrix} = \operatorname{diag}[1,1,2]^{-1}\,\mathbf{J}\,\operatorname{diag}[2,2,1].$$

[1] In practice, storing a zero vector is not necessary; one can store a flag indicating that the gradient is zero.

7 Concluding remarks.

Given a DAE described by a computer program, we have shown how the necessary structural analysis data and System Jacobian can be readily obtained via operator overloading, as in DAETS; or how they can be generated through source-text translation.

The single-pass method is expected to be much more efficient than an operator-overloading approach for computing the System Jacobian. However, the former has not been implemented yet, while the latter is incorporated into DAETS without major obstacles.

The techniques described in this paper can be used on their own to perform SA of a DAE: if $\mathbf{J}$ is nonsingular at a consistent point, within round off, then the DAE is solvable in a neighbourhood of this point. By results in [2, 4], we can determine the structural index of the DAE, which is an upper bound on its differentiation index, and we can also find the degrees of freedom of the DAE. Moreover, simulation software that automatically generates the equations of a DAE system need not produce them in a particular (first-order or lower-index) form: they can be a direct, compact translation of the model.

REFERENCES

1. A. Griewank, *On automatic differentiation*. In Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, Dordrecht, 1989, pp. 83–108.
2. N. S. Nedialkov and J. D. Pryce, *Solving differential-algebraic equations by Taylor series (I): Computing Taylor coefficients*. BIT, Accepted 2005.
3. C. C. Pantelides, *The consistent initialization of differential-algebraic systems*. SIAM. J. Sci. Stat. Comput., 9 (1988), pp. 213–231.
4. J. D. Pryce, *A simple structural analysis method for DAEs*. BIT, 41 (2001), pp. 364–394.
5. O. Stauning and C. Bendtsen, *FADBAD++ web page*, May 2003. FADBAD++ is availabe at `www.imm.dtu.dk/fadbad.html`.