On the Efficient Generation of Taylor Expansions for DAE Solutions by Automatic Differentiation

Andreas Griewank[1] and Andrea Walther[2]

[1] Department of Mathematics, Humboldt-Universität Berlin, Germany, griewank@mathematik.hu-berlin.de.
[2] Institute of Scientific Computing, Technische Universität Dresden, Germany, awalther@math.tu-dresden.de.

Abstract. Under certain conditions the signature method suggested by Pantiledes and Pryce facilitates the local expansion of DAE solutions by Taylor polynomials of arbitrary order. The successive calculation of Taylor coefficients involves the solution of nonlinear algebraic equations by some variant of the Gauss-Newton method. Hence, one needs to evaluate certain Jacobians and several right hand sides. Without advocating a particular solver we discuss how this information can be efficiently obtained using ADOL-C or similar automatic differentiation packages.[3]

1 Introduction

The differential algebraic systems in question are specified by a vector function

$$F(t, \mathbf{y}) \equiv F(t, \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n) \quad \text{with}$$
$$F : \mathbb{R}^{1+m} \equiv \mathbb{R} \times \mathbb{R}^{1+m_1} \times \mathbb{R}^{1+m_2} \times \cdots \times \mathbb{R}^{1+m_n} \mapsto \mathbb{R}^n \ .$$

Here, t denotes the independent "time" variable, n is the dimension of the state space and the $\mathbf{y}_j$ are $(1 + m_j)$-dimensional vectors whose rth component $y_{j,r}$ represents the rth derivative of the state space component $y_j \equiv \mathbf{y}_{j,0}$. However, this relation is really of no importance as far as the pure automatic differentiation task is concerned.

In principle, F can be an arbitrary algebraic mapping from $\mathbb{R}^{1+m}$ to $\mathbb{R}^n$ but for our purposes it should be defined by an evaluation procedure in a high level computer language like Fortran or C. Then, the technique of automatic differentiation (AD) offers an opportunity to provide derivative information of any order for the given code segment by applying the chain rule systematically to statements of computer programs. For that purpose, the code is decomposed into a typically very long sequence of simple evaluations, e.g. additions, multiplications, and calls to elementary functions such as $\sin(x)$ or $\exp(x)$. The derivatives with respect to the arguments of these operations can be easily calculated. Exploiting the chain rule yields the derivatives of the whole sequence of

[3] This work was supported by the DFG research center MATHEON, Mathematics for Key Technologies in Berlin, and DFG grant WA 1607/2-1

statements with respect to the input variables. Depending on the starting point of this methodology—either at the beginning or at the end of the chain of computational steps—one distinguishes between the forward mode and the reverse mode of AD. Using the forward mode, one computes the required derivatives together with the function evaluation in one sweep. Applying the reverse mode of AD, the derivative values are propagated during a backward sweep. Hence after a function evaluation, one starts computing the derivatives of the dependents with respect to the last intermediate values and traverses backwards through the evaluation process until the independents are reached. A comprehensive exposition of these techniques of AD can be found in [7].

For many DAEs the computational graph related to the code segment to evaluate $F(t, \mathbf{y})$ will be of quite manageable size, but still we will try to keep the number of sweeps through it as low as possible. Moreover, we will try to amortize the overhead of each sweep by performing reasonably intense calculations for each graph vertex, which represents an intermediate quantity in the evaluation of the algebraic function $F(t, \mathbf{y})$.

The structure of the paper is the following. In Section 2 we discuss the efficient computation of first and higher-order derivatives using automatic differentiation. The structural analysis required for the signature method suggested by Pantelides [9] and Pryce [10, 11] is the subject of Section 3. New drivers of the AD-tool ADOL-C [8] are presented for the first time in Section 4. They are tailored especially for the use in the DAE context. Preliminary numerical results illustrating the effort to compute higher-order derivatives are discussed in Section 5. Section 6 contains some conclusions and an outlook.

2 Computing First and Higher Order Derivatives

Throughout the paper we assume that the time t has been shifted so that its current value is simply $t = 0$. Then we obtain for any analytic path

$$\mathbf{y}(t) = \sum_{r=0}^{\bar{r}} \mathbf{y}^{(r)} t^r$$

a corresponding value path

$$F(t, \mathbf{y}(t)) = \sum_{r=0}^{\bar{r}} t^r \hat{F}_r(0, \mathbf{y}^{(0)}, \mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(r)}) + \mathcal{O}(t^{\bar{r}+1})$$

The coefficient functions $\hat{F}_r : \mathbb{R}^{1+m*r} \mapsto \mathbb{R}^n$ are analytic provided this is true for F as we assume for simplicity. To compute the desired higher-order information, we will first examine the derivative computation for intrinsic function. For a given Taylor polynomial

$$x(t) = x_0 + x_1 t + x_2 t^2 + \ldots + x_{\bar{r}-1} t^{\bar{r}} \in \mathbb{R}^n$$

a naive implementation to compute higher-order derivatives could be based on "symbolic" differentiation. This approach would yield for a general smooth $v(t) = \varphi(x(t))$ the derivative expressions

$$\begin{aligned}
v_0 &= \varphi(x_0) \\
v_1 &= \varphi_1(x_0)\, x_1 \\
v_2 &= \varphi_2(x_0)\, x_1\, x_1 + \varphi_1(x_0)\, x_2 \\
v_3 &= \varphi_3(x_0)\, x_1\, x_1\, x_1 + 2\,\varphi_2(x_0)\, x_1\, x_2 + \varphi_1(x_0)\, x_3 \\
v_4 &= \varphi_4(x_0)\, x_1\, x_1\, x_1\, x_1 + 3\varphi_3(x_0)\, x_1\, x_1\, x_2 \\
&\quad +\varphi_2(x_0)\, (x_2\, x_2 + 2\, x_1\, x_3) + \varphi_1(x_0)\, x_4 \\
v_5 &= \quad \ldots
\end{aligned}$$

Hence, the overall complexity grows rapidly in the highest degree $\bar{r}$ of the Taylor polynomial. To avoid these prohibitively expensive calculations the standard higher-order forward sweep of automatic differentiation is based on Taylor arithmetic [2] yielding an effort that grows like $\bar{r}^2$ times the cost of evaluating $F(t, \mathbf{y})$. This is quite obvious for arithmetic operations as shown below. For a general elemental function φ, one finds also a recursion with quadratic complexity by interpreting φ as solution of a linear ODE. The following table illustrates the resulting computation of the Taylor coefficients for a simple multiplication and the exponential function:

$v(t) =$	Recurrence for $k = 1 \ldots \bar{r} - 1$	OPS	MOVES
$x(t) * y(t)$	$v_k = \sum_{j=0}^{k} x_j y_{k-j}$	$\sim \bar{r}^2$	$3\,\bar{r}$
$\exp(x(t))$	$k v_k = \sum_{j=1}^{k} j v_{k-j} x_j$	$\sim \bar{r}^2$	$2\,\bar{r}$

Similar formulas can be found for all intrinsic functions. This fact permit the computation of higher-order derivatives for the vector function $F(t, \mathbf{Y})$ as composition of elementary components. The AD-tool ADOL-C [8] uses the Taylor arithmetic as described above to provide an efficient calculation of higher-order derivatives. Furthermore, the AD-tools FADBAD [1] and CppAD [4] use the same approach to compute higher-order information.

In ODE and DAE solving, the coefficients $\mathbf{y}^{(r)}$ are generated in increasing order using successive values of the residuals $\hat{F}_r$. For that purpose, we have to compose the input coefficients of the vectors

$$\mathbf{y}^{(r)} = [\mathbf{y}_1^{(r)}, \mathbf{y}_2^{(r)}, \ldots, \mathbf{y}_n^{(r)}]$$

from the values $y_{j,s}$ obtained so far. This simple application of the chain rule is the only extra procedure we have to attach to our AD software to facilitate the calculation of the desired Taylor coefficients. Specifically, using ADOL-C we must set

$$y_{j,s}^{(r)} = y_{j,r+s}/s!$$

because the computations performed by ADOL-C are based on the unscaled Taylor coefficients.

If the $\hat{F}_s$ for $s \le r$ are reevaluated from scratch every time this requires r sweeps and thus a computational effort of order r^3 times the cost of evaluating the underlying algebraic mapping $F(t, \mathbf{y})$. As observed in [7, Section 10] there are at least two ways in which this effort can be reduced to being quadratic in r. The first option is to store and retrieve the partial Taylor polynomials of all intermediate quantities that occur during the evaluation of F. This has been done in the Fortran package ATOMFT [5] for the solution of ODEs by the Taylor series method.

The second possibility is to exploit the property that F_r is linear in all $\mathbf{y}^{(s)}$ with $s > r/2$ so that in fact

$$\begin{aligned} &F_r(0, \mathbf{y}^{(0)}, \mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(r)}) \\ &= F_r(0, \mathbf{y}^{(0)}, \ldots, \mathbf{y}^{(s-1)}, 0, \ldots 0) + \sum_{k=s}^{r} A_{k-s}(0, \mathbf{y}^{(0)}, \ldots, \mathbf{y}^{(k-s)}) \mathbf{y}^{(s)} \end{aligned}$$

Here the $A_s(0, \mathbf{y}^{(0)}, \ldots, \mathbf{y}^{(k-s)}) \in \mathbb{R}^{n \times n}$ for $s \le r$ are the Taylor coefficients of the Jacobian path $J(0, \mathbf{y}(t))$. They can also be evaluated by standard AD methods and are needed anyway if one wishes to compute sensitivities of the Taylor coefficients with respect to the basis point $\mathbf{y}^0$ in an implicit Taylor method. In contrast to the save and restore option, exploiting the linearity reduces the number of sweeps through the computational graph essentially to the logarithm of the maximal order $\bar{r}$.

3 Structural Analysis in terms of the Jacobian J

The structural analysis used by Pantelides [9] has become part of professional simulation software [3, 6] and has been applied successfully to a wide variety of systems. Nevertheless, it has to be mentioned, that Pantelides' algorithm applied to DAEs of index 1 may perform an arbitrarily high number of iterations and differentiations [12]. This behaviour is due to the fact that the structural index of the DAE may exceed the index of the DAE and that the differentiation needed by Pantelides' algorithm relate to the structural index. However, the present paper focuses on the derivative computation. Therefore, these possibly difficulties are just a side note for computing the consisted point. They might be overcome in the future by a better suited determination method for the negative shifts mentioned below.

In the structural analysis used by Pantelides [9] and Pryce [10, 11], the elements σ_{ij} of the signature matrix Σ are defined by

$$\sigma_{ij} = \begin{cases} \max\{r \mid y_{j,r} \text{ occurs in } F_i\} \\ -\infty \quad \text{if no component of } \mathbf{y}_j \text{ occurs in } F_i \end{cases},$$

where F_i denotes the ith component function of F. Here, "$y_{j,r}$ occurs in F_i" means that the value of the latter depends nontrivially on the former, which

leads to its occurrence in a symbolic expression for F_i. The signature matrix is used to determine vectors $c = (c_i)_{i=1}^n$ and $d = (d_j)_{j=1}^n$ of nonnegative shifts. It is shown in [10] that if a solution

$$\mathbf{Y}^* = (y^*_{1,0}, \ldots, y^*_{1,d_1}, \ldots, y^*_{n,0}, \ldots, y^*_{n,d_n})$$

of the equations

$$\left.\begin{array}{c} F_1^{(0)}, F_1^{(1)}, \ldots, F_1^{(c_1)} \\ \vdots \\ F_n^{(0)}, F_n^{(1)}, \ldots, F_n^{(c_n)} \end{array}\right\} = 0$$

exists and the system Jacobian J with

$$J_{ij} = \begin{cases} \dfrac{\partial F_i}{\partial y_{j,d_j - c_i}} & \text{if } y_{j,d_j-c_i} \text{ occurs in } F_i \\ 0 & \text{otherwise} \end{cases}$$

is nonsingular at the solution, then $\mathbf{Y}^*$ is a consistent point of the DAE at time t. The required derivative information for constructing J can be obtained by a single reverse sweep in vector mode to evaluate the rectangular Jacobian

$$\begin{aligned} J(0, \mathbf{y}) &\equiv [J_1(0, \mathbf{y}), J_2(0, \mathbf{y}), \ldots, J_n(0, \mathbf{y})] \in \mathbb{R}^{n \times m} \qquad \text{with} \\ J_j(0, \mathbf{y}) &\equiv \frac{\partial F(0, \mathbf{y})}{\partial \mathbf{y}_j} \in \mathbb{R}^{n \times (1 + m_j)} \, . \end{aligned}$$

Irrespective of the size of m, the operations count for this will be about n times that of evaluating $F(t, \mathbf{y})$ by itself.

To compute a solution $\mathbf{Y}^*$ one has to solve a sequence of underdetermined systems of nonlinear equations. For that purpose, one needs the corresponding Jacobian. This matrix is given by a part of the system Jacobian and can be evaluated using again either standard higher-order automatic differentiation or the more efficient variants discussed above. However, the initialization of the input Taylor coefficients is considerably easy since one simply has to choose the corresponding unit vectors.

Once a consistent point at time $t = 0$ is computed, one may for example apply an explicit Taylor method to integrate the DAE. For that purpose, only one solve of a linear system with the system Jacobian J as linear operator is required for each order of the Taylor method. Hence, one performs one LU-factorization of J. Subsequently, one only has to evaluate higher-order derivatives occurring in the right-hand sides. For this purpose, the technique explained in the preceding section can be used.

4 Implementation details

For simplicity we have assumed that the DAE system has been written in autonomous form, but an explicit time dependence could certainly be accounted for

too. Although variations are possible we suggest that the problem be specified by an evaluation code of the following form

```
void sys_eval(int n, adouble** y, adouble* F)
```

using the active variable type adouble provided by ADOL-C. Here y[j][k] represents $y_{j,k}$, i.e. the k-th derivative of the j-th variable with $k \leq m_j$. In other words, the calling program must have allocated n adouble pointers y[j] for $j = 0, \ldots, n-1$ where each of them is itself a vector of $m_j + 1 =:$ m[j] adoubles. On exit the components F[i] for $i = 0, \ldots, n-1$ contain the function components F_i in Pryce notation.

Provided the code sys_eval does not contain any branches it must be called only once before the actual DAE solving begins. Before the call, y and F must be allocated and y initialized by a loop of the form

```
int tag = 1;
trace_on(tag)
  y = new adouble*[n]; F = new adouble[n];
  for(j=0; j<n; j++)
  {  y[j] = new adouble[m[j]];
     for (i=0; i<m[j]; i++)
       y[j][i]<<=yp[j][i];
  }
  sys_eval(n, y, F)
  for(j=0; j<n; j++)
    F[j] >>= Fp[j]
trace_off()
```

Here, yp[j][i] is an array of double values at which sys_eval can be sensibly called. The loop after the call to sys_eval determines the dependent variables in ADOL-C terminology, where Fp[j] is like yp[j][i] of type double. Now the DAE system has been *taped*. If the function evaluation contain branches, the generated tape can be reused as long as the control does not change. If the control flow changes, the return values of the drivers for computing the desired derivatives indicate that the tape is no longer valid and a retaping has to be performed. Hence, by monitoring the corresponding return values the correctness of the derivative information can be ensured while keeping the effort for the taping as low as possible.

Once, the tape is generated, function and derivative evaluations are performed for example by the routines zos_forward_partx(..), fos_forward_partx(..), hos_forward_partx(..), and jacobian_partx(..) that are problem independent. For example the call

```
zos_forward_partx(tag,n,n,m,yp,Fp)
```

will yield as output the system values Fp[j] for arbitrary inputs yp[j][i] and the array m describing the partition of yp. Here, zos_forward stands for **z**ero-**o**rder **s**calar forward mode, since no derivatives are required.

Now suppose we have allocated and assigned values to a three dimensional tensor y[j][i][r] for $j < n, i < m[j], r \leq b$. Mathematically, this is interpreted as the family of Taylor expansions

$$\mathsf{y[j][i]} \equiv \sum_{r=0}^{b} \mathsf{yt[j][i][r]}\ t^r \quad .$$

For ADOL-C, the values y[j][i][r] are completely independent but for use in the integration method proposed by Pryce they must be given values that are consistent in that

$$\mathsf{yt\ [j][i][r]} = y_{j,i+r}/r! \quad .$$

Here the $y_{j,i+r} = y_i^{(i+r)}$ are the already known or guessed solution values and derivatives. All derivatives of higher-order should be set to zero. Now the call

```
fos_forward_partx(tag,n,n,m,yt,Ft)
```

yields the Taylor coefficients Ft[j][r] for $r < 2$, where fos_forward stands for **f**irst-**o**rder **s**calar forward mode and the call

```
hos_forward_partx(tag,n,n,m,b,yt,Ft)
```

yields the Taylor coefficients Ft[j][r] for $r \leq b$ of the resulting expansion

$$\mathsf{F[i]}\ \equiv \sum_{r=0}^{b} \mathsf{Ft[i][r]}\ t^r.$$

In other words, the derivative $(r!)$ Ft[j][r] corresponds exactly to the value $F_{j,r}$ needed for the approach described in [10, 11] by Pryce, except for the scaling by $r!$ that has to be done by the user.

For computing the system Jacobian that is also required by the method of Pryce and similar integration methods, ADOL-C provides also a special driver. For using it one must allocate the array jac[i][j][k] for $i < n, j < n, k \leq m_j$. In order to obtain the values

$$\mathsf{jac[i][j][k]} \equiv \partial F_i/\partial y_{j,k}$$

of this Jacobian, the user has to call the new Jacobian driver

```
jacobian_partx(tag,n,m,n,xp,jac).
```

of ADOL-C. The presented new driver of ADOL-C are available in the current version 1.9.0 and have been incorporated into a software-prototype in order to test the calculation of higher order derivatives for the integration of high-index DAEs. The achieved numerical results are presented in the next section.

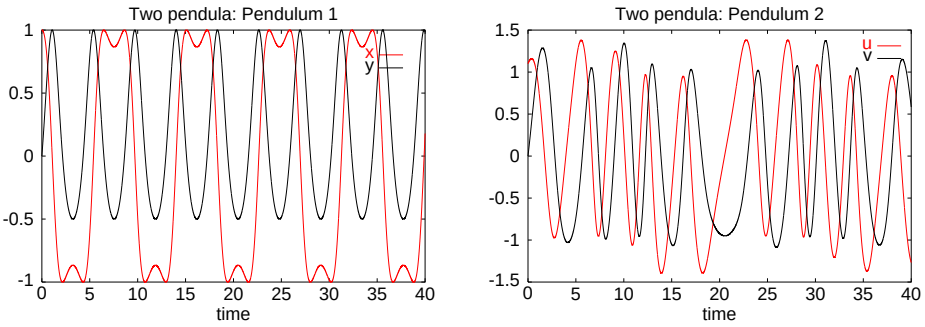

Fig. 1. Index 5 two-pendulum problem, numerical results

5 Numerical example

We implemented a very simple version of the algorithm given in [10] to illustrate the capabilities of ADOL-C to provide the required higher-order derivatives. Furthermore, the resulting code may form one possibility to verify the run-time saving that can be achieved using the improvements stated in Section 2.

As test example, we choose a model of two pendula from [10], where the λ component of the first one controls the length of the second one. This system with index 5 is described by the DAE

$$\begin{aligned} F_1 &= x'' + x\lambda &&= 0 & \qquad F_4 &= u'' + u\kappa &&= 0 \\ F_2 &= y'' + y\lambda - g &&= 0 & \qquad F_5 &= v'' + v\lambda - g &&= 0 \\ F_3 &= x^2 + y^2 - L &&= 0 & \qquad F_6 &= u^2 + v^2 - (L + c\lambda)^2 &&= 0 \end{aligned}$$

and has four degrees of freedom. For the numerical tests presented below, we use the gravity constant $g = 1$, the length $L = 1$ of the first pendulum, $c = 0.1$ and simulate the behaviour of both pendula for the time interval $[0, 40]$. This choice of T ensures that we simulate the pendula for a period where they do not show chaotic behaviour, which is the case for $T \geq 50$, cf. [10]. The values $(x_0, y_0) = (u_0, v_0) = (1, 0)$ and $(x_0', y_0') = (u_0', v_0') = (0, 1)$ serve as initial point. In order to judge the influence of the derivative calculation on the run-time, we consider three discretizations based on the step size $h = 0.0025, 0.005, 0.01$ which results in the time step numbers 16 000, 8 000 and 4 000 respectively. In addition we use explicit Taylor methods of order $5, 10, 15, 20, 25, 30$ for the integration of the DAE.

The computed results for the two pendula are illustrated by Fig. 1, where the first one shows obviously a periodic behaviour. Since the length of the second one varies in dependence on the first pendulum one can observe for the second one a non-periodic solution which results eventually in a chaotic behaviour. However, for the chosen combinations of step size and integration order the computed solutions are identical for the first pendulum and close or at least comparable for the second one. This fact was acceptable for us since the influence of the derivative order on the computing time was the main subject.

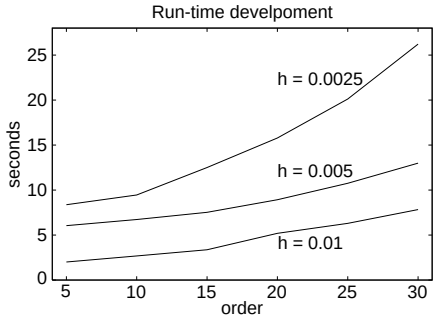

Fig. 2. Index 5 two-pendulum problem, run-times

The simulations were computed using a Red Hat Linux system, with an AMD Athlon XP 1666 Mhz processor and 512 MB RAM. The required computing times for the different step sizes and integration orders are illustrated by Fig. 2. Here, for one specific step size the computational efforts varies mainly due to the integration order of the explicit Taylor method. The predicted nonlinear behaviour can be observed very clearly for the step size $h = 0.0025$. For the larger time steps, i.e. $h = 0.01$ and $h = 0.005$, the derivative calculation is dominated by the linear algebra cost. Therefore, Fig. 2 shows only a slight nonlinear influence of the integration order on the run-time. Nevertheless, the numerical experiment confirms that the effort for computing higher-order derivatives increases only moderately when using the AD-tool ADOL-C. This is in accordance to the theoretical results sketched in Section 2.

6 Conclusion and Outlook

This paper discusses the computation of higher-order derivatives using automatic differentiation in the context of high index DAEs. For that purpose, the standard higher-order forward sweep of automatic differentiation based on Taylor arithmetic is discussed as well as two possible improvements that are valuable for very high order derivatives. This methodology is embedded in the structural analysis for high-index DAEs. One case study presents numerical results achieved with the AD-tool ADOL-C that confirm the theoretical complexity of computing higher-order derivatives.

Certainly, the implementation of the improvements for the generation of Taylor expansions presented in this paper forms one specific future challenge. This may include also a possibly thread-based parallelization. Here one can exploit the coarse-grained nature of Taylor computations that differ significantly from the situation when computing first-order derivatives using AD. An additional task is to ease the use of existing AD-tools for the usage in connection with the structural approach to integrate high-index DAEs. For that purpose, we presented a description of new drivers provided by ADOL-C 1.9.0 that take into

account the special structure of a given DAE system where in addition to the values of the variables also the values of specific derivatives of the variables enter the evaluation of the system function.

References

1. C. Bendtsen and O. Stauning: FADBAD, a flexible C++ package for automatic differentiation. Department of Mathematical Modelling, Technical University of Denmark, 1996.
2. R. Brent and H. Kung: Fast algorithms for manipulating formal power series. Journal of the Association for Computing Machinery **25**, 581–595, 1978.
3. F. Cellier and H. Elmquist: Automated formula manipulation supports object-oriented continuous-system modelling. IEEE Control System Magazine **13**, 28–38, 1993.
4. http://www.seanet.com/~bradbell/CppAD/
5. Y.F. Chang and G. Corliss: Solving ordinary differential equations using Taylor series. ACM Trans. Math. Software **8**, 114–144, 1982.
6. W. Feehery and P. Barton: Dynamic optimization with state variable path constraints. Comput. Chem. Engrg. **22**, 1241–1256, 1998.
7. A. Griewank: *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*, Frontiers in Appl. Math. 19, SIAM, Phil., 2000.
8. A. Griewank, D. Juedes, and J. Utke: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. TOMS **22**, 131–167, 1996.
9. C.C. Pantelides: The consistent initialization of differential-algebraic systems. SIAM J. Sci. Statist. Comput. **9**, 213–231, 1988.
10. J. Pryce: Solving high-index DAEs by Taylor series. Numer. Algorithms **19**, 195–211, 1998.
11. J. Pryce: A simple structural analysis method for DAEs. BIT **41**, 364–394, 2001.
12. G. Reißig, W. Martinson, and P. Barton: Differential-algebraic equations of index 1 may have an arbitrarily high structural index. SIAM J. Sci. Comput. **21**, 1987–1990, 2000.