`daeIndexDet`: A Program for the Index Determination in DAEs using the `indexdet` Library and the ADOL-C Package for Automatic Differentiation

Dagmar Monett
DFG Research Center MATHEON
Institute of Mathematics
Humboldt-Universität zu Berlin
Unter den Linden 6,
10099 Berlin, Germany
monett@math.hu-berlin.de

July 5, 2007

Abstract

This document explains how to use a program for index determination in systems of differential algebraic equations by using Automatic Differentiation techniques. It will lead the users through the steps needed from creating their examples (e.g. DAEs) to finally running the program that computes their index.

Contents

1 Introduction **2**
- 1.1 Index determination in DAEs 2

2 Getting Started **3**
- 2.1 The distribution files 4
  - 2.1.1 Installing the libraries 4
- 2.2 Compiling and linking the program 5
- 2.3 Running the program 5

3 Defining the DAE, the Dynamic, and the Trajectory **6**
- 3.1 User example classes 6

4 Creating and Referencing Objects of the User Classes **8**

5 Parameters Controlling the Program Run **9**
- 5.1 The degree of Taylor coefficients 9
- 5.2 The QR and IO thresholds: Controlling the precision 9
  - 5.2.1 The QR threshold 9
  - 5.2.2 The IO threshold 10
- 5.3 The printout parameters: Controlling the information to be printed out 10
  - 5.3.1 What to print 10
  - 5.3.2 Where to print 10

6 Default Settings **10**
- 6.1 Modifying the default settings 12
  - 6.1.1 Modifying the values one by one 12
  - 6.1.2 Modifying several values at a time 13

7 Main Function Call **13**

8 All in All **14**

9 Automatic Differentiation using ADOL-C **15**
- 9.1 Active Sections: Computing the Taylor coefficients 15
  - 9.1.1 Active section to compute the trajectory 15
  - 9.1.2 Active section to compute the dynamic 17
  - 9.1.3 Active section to compute the DAE 18
  - 9.1.4 Constructing the matrices A, B, and D 19

References **20**

1 Introduction

daeIndexDet stands for *Index Determination in DAEs* and is a program for computing the index in systems of differential algebraic equations (DAEs) with properly stated leading terms. It uses the indexdet library, which actually provides the functionalities for the index computation. Its current version is coded in C++.

The main strengths of the indexdet library we introduce are:

- Index calculation based on a matrix sequence with suitable chosen projectors by using Automatic Differentiation (AD) techniques [1, 2].
- Evaluation of derivatives using the C++ package ADOL-C [7]. ADOL-C provides easy-to-use drivers that compute convenient derivative evaluations with only few modifications to the original C++ code.
- Inclusion of classes for implementing Taylor arithmetic functionalities. Basic operations with and over both Taylor polynomials and matrices of Taylor polynomials, as well as Linear Algebra functions (several matrix multiplications and the Householder QR factorization with column pivoting being the most relevant ones) that complement the index determination are provided by overloading built-in operators in C++.
- Extension of the exception handling mechanisms from C++ to provide a robust solution to the shortcomings of traditional error handling methods, like those that may occur during execution time when computing the index.

1.1 Index determination in DAEs

We deal with DAEs given by the general equation

$$f((d(x(t),t))', x(t), t) = 0, \qquad t \in I \tag{1}$$

with $I \subseteq \mathbb{R}$ being the interval of interest.

How to compute the index of these DAEs is addressed in [4, 5, 6]. Special attention is paid to the coefficients

$$A(t) := \left(\frac{\partial f}{\partial z}\right), \qquad B(t) := \left(\frac{\partial f}{\partial x}\right), \qquad \text{and} \qquad D(t) := \left(\frac{\partial d}{\partial x}\right)$$

with $z(t) = d'(x(t), t)$.

The matrices $A(t) \in \mathbb{R}^{n \times m}$, $B(t) \in \mathbb{R}^{n \times n}$, and $D(t) \in \mathbb{R}^{m \times n}$ are supposed to be continuous. Further, the coefficients $A(t)$ and $D(t)$ should be well matched, which makes the DAE's leading term properly stated. The computation of the index is based on a matrix sequence given the coefficients $A(t)$, $B(t)$, and $D(t)$.

By forming the sequence of matrices, suitable chosen projectors are computed using generalized inverses.

In [3], an algorithm is proposed to realize the matrix sequence and to finally compute the index. Our `indexdet` library implements such an algorithm with only some modifications, the most relevant ones being the way the matrices $A(t)$, $B(t)$, and $D(t)$ are computed (via ADOL-C), as well as how the time differentiations in the matrix sequence are done (via a shift operator over the Taylor series).

In the rest of this article we name $d(x(t), t)$ and $x(t)$ from Equation 1 the *dynamic* and the *trajectory*, respectively.

The DAE, the dynamic, and the trajectory are defined as follows:

- The function $f : \mathbb{R}^{m_{dyn}} \times \mathbb{R}^{m_{tra}} \times \mathbb{R} \to \mathbb{R}^{m_{dae}}$ defines the DAE, $n_{dae} = m_{dyn} + m_{tra} + 1$ being the number of independent variables and m_{dae} being the number of dependent variables. The independent variables of the DAE $f(z(t), x(t), t) = 0$, with $z(t) = d'(x(t), t)$ are $z(t)$, $x(t)$ and t.

- The function $d : \mathbb{R}^{m_{tra}} \times \mathbb{R} \to \mathbb{R}^{m_{dyn}}$ defines the dynamic, $n_{dyn} = m_{tra} + 1$ being the number of independent variables and m_{dyn} being the number of dependent variables. The function $d(x(t), t)$ has two independent variables, $x(t)$ and t, that are previously computed at the time of computing $d(x(t), t)$.

- The function $x : \mathbb{R} \to \mathbb{R}^{m_{tra}}$ defines the trajectory, m_{tra} being the number of dependent variables. The function $x(t)$ depends on the independent variable t, which value is given by the user.

2 Getting Started

The steps needed to compute the index by using the `indexdet` library are grouped into the following:

1. Installing the distribution files and libraries.

2. Creating and editing the user header where the DAE and other related information are defined.

3. Editing the `daeIndexDet` C++ source file that computes the index for the already defined user example.

4. Compiling and linking the `daeIndexDet` C++ source file.

5. Running the `daeIndexDet` program.

The next sections will explain these steps in detail.

2.1 The distribution files

Figure 1 shows a general schema with the most important libraries and files that are needed for the index determination.

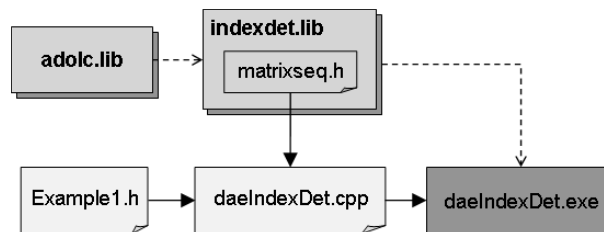

Figure 1: The main libraries and files needed for the index determination.

The libraries `adolc.lib` and `indexdet.lib` provide the functionalities for automatic differentiation and index determination, respectively. We use the `adolc.lib` library for the evaluation of derivatives using the C++ package ADOL-C. We introduce the `indexdet.lib` library for the index calculation based on a matrix sequence with suitable chosen projectors.

2.1.1 Installing the libraries

Both the `adolc.lib` and `indexdet.lib` libraries should be previously installed and properly declared in the user path.

The ADOL-C package, which includes the `adolc.lib` library, can be downloaded from the ADOL-C's web site [7]. We suggest to follow the steps described in the files `INSTALL` and `README` there provided where proper instructions for installing ADOL-C on both *NIX and Windows platforms are documented.

For example, on *NIX platforms we have done:

```
$ ./configure --with-docexa --with-addexa
$ make
$ make install
$ make clean
```

which configure, compile, and install the package's files on `{HOME}/adolc_base/` (the location by default) thereby including documented and additional examples.

The files for the `indexdet.lib` library, by now, are provided together with the program `daeIndexDet` in the file `indexdet.tar.gz`. First, copy the file `indexdet.tar.gz` on the location of your preference. Then, on *NIX machines type:

```
$ tar zxvf indexdet.tar.gz
```

for decompressing the file. This instruction generates the local subdirectory `indexdet` containing, in the current version, all needed files for both the program and the library.

The provided `makefile` might be edited by the user (see next section for details). It allows for compiling and linking both the program `daeIndexDet` and the library `indexdet.lib`.

The C++ file `daeIndexDet.cpp` might be also edited by the user (see Section 4 for details). It allows for computing the index of the DAE, typed in the header `Example1.h` (see Section 3) or any other user example.

2.2 Compiling and linking the program

When necessary, the `makefile` provided with the `daeIndexDet` program related files should be edited. Its variable `ADOLCDIR` defines the subdirectory where the ADOL-C package is installed. The ADOL-C subdirectory is assumed to be `(HOME)/adolc_base` by default. The user should modify this location conveniently.

Edit also the program `daeIndexDet.cpp` properly, according to the DAE for which the index should be computed.

For ensuring compatibility, we recommend to use the same C/C++ compiler for all needed compilations (e.g. when compiling the libraries `indexdet.lib` and `adolc.lib`, and the header defining the DAE).

For compiling and linking the program `daeIndexDet` on *NIX machines type:

```
$ ./makefile
```

from the already created local subdirectory `indexdet`. The linking process should generate the program `daeIndexDet.exe` on the same location.

2.3 Running the program

The `daeIndexDet` program can be executed by typing its name at the system prompt. On *NIX machines simply type:

```
$ ./daeIndexDet
```

On Windows machines type instead:

```
> daeIndexDet
```

3 Defining the DAE, the Dynamic, and the Trajectory

The user header (i.e. `Example1.h` in Figure 1) contains the DAE for which its index should be computed. In that header, the user should define not only the DAE but also the dynamic and the trajectory. In other words, the user should provide the functions $x(t)$, $d(x(t), t)$, and $f(z(t), x(t), t)$ in a separate C++ class. The structure of this class will be presented in the following.

3.1 User example classes

The definitions for the functions $x(t)$, $d(x(t), t)$, and $f(z(t), x(t), t)$ should be provided by the user. She or he can realize this through a self implemented class that should be compiled before the program to determine the index is run.

Such a user class inherits a general structure and diverse functionalities from the abstract base class `IDExample` that is coded in the C++ header `IDExample.h` (provided in the library `indexdet.lib`). Instances of the class `IDExample` can not be created: it is an *abstract class*. However, pointers to the derived user classes are type-compatible with a pointer to the base class, which makes polymorphism a powerful feature that is exploited by the program to determine the index.

Let the trajectory $x(t)$ be defined by

$$x_1(t) = \sin t, \tag{2}$$
$$x_2(t) = \cos t, \tag{3}$$
$$x_3(t) = \log 1 + t, \tag{4}$$

For this example, $m_{tra} = 3$ since $x(t)$ is defined by three equations that depend on the independent variable t. Its C++ code in the header `Example1.h` corresponds to the following instructions:

```
void tra( adouble t, adouble *ptra )
{
  ptra[ 0 ] = sin( t );
  ptra[ 1 ] = cos( t );
  ptra[ 2 ] = log( 1 + t );
}
```

`t` being the `adouble` independent variable and `ptra` being a pointer to an `adouble` vector defining the trajectory.

Let the dynamic $d(x(t), t)$ be defined by

$$d_1(x(t), t) = -2 \cdot \sqrt{1 - x_1(t)}, \tag{5}$$
$$d_2(x(t), t) = \sin t \cdot x_2(t), \tag{6}$$

In this case, $m_{dyn} = 2$ and $n_{dyn} = 4$, since $d(x(t), t)$ is defined by two equations that depend on four independent variables (i.e. three from $x(t)$ and t).

Finally, let the DAE $f(z(t), x(t), t)$ be defined by

$$f_1(t) = z_1(t) + x_3(t), \tag{7}$$
$$f_2(t) = z_2(t) - x_3(t), \tag{8}$$
$$f_3(t) = x_3(t) - g(x_1(t) - x_2(t)), \tag{9}$$

with $z(t) = d'(x(t), t)$ and $g(x) = e^x - 1$. Here, $m_{dae} = 3$ according to the number of equations that define $f(z(t), x(t), t)$, and $n_{dae} = 6$ because we have three independent variables from $x(t)$, two from $z(t)$, and the independent variable t.

The user should code all these equations (i.e. her/his class example) in the already mentioned C++ header file. For example, the following code shows the user class `Example1.h` and its member functions. It is a polymorphic class because it inherits virtual functions from the abstract base class `IDExample`.

```
/** File Example1.h */
#ifndef EXAMPLE1_H_
#define EXAMPLE1_H_
#include "IDExample.h"    // Abstract base class.

class Example1 : public IDExample {
    public:
        /**
         * Class constructor.
         */
        Example1( void ) : IDExample( 2, 3, 0.0, "Example1" )
        { }

        /**
         * Definition of the trajectory x(t).
         */
        void tra( adouble t, adouble *ptra )
        { //... }

        /**
         * Definition of the dynamic d(x(t),t).
         */
        void dyn( adouble *px, adouble t, adouble *pd )
        { //... }
```

```
        /**
         * Definition of the DAE f(z(t),x(t),t).
         */
        void dae( adouble *py, adouble *px, adouble t, adouble *pf )
        { //... }
};

#endif /*EXAMPLE1_H_*/
```

Notice that the class constructor for the user class `Example1` does not have any parameter. However, it calls the class constructor of the abstract base class `IDExample` with specific ones:

1. The first parameter corresponds to the number of dependent variables of the dynamic $d(x(t), t)$ and its type is `int` (equal to 2 in the example). It must be equal to the number of equations that define the dynamic.
2. The second parameter corresponds to the number of dependent variables of the DAE $f(z(t), x(t), t)$. Its type is also `int`. Its value must be equal to the number of equations that define the DAE, as well as be equal to the number of equations that define the trajectory $x(t)$ (3 in the example).
3. The third parameter is a `double` that indicates the point at which the index should be computed, i.e., the value for the independent variable time ($t_0 = 0.0$ in the example).
4. The fourth and last parameter is a character string used to denote the output files with numerical results. For example, the name of the class can be used ("$Example1$" in the example).

This is the only information, besides the definition of the trajectory, the dynamic, and the DAE, that is required from the user.

When an object of type `Example1` is created, there are initialized also other parameters related to the example as well, as those corresponding to the resting dimensions already mentioned. The functionality of $x(t)$, $d(x(t), t)$, and $f(z(t), x(t), t)$ should be coded in the member functions `tra`, `dyn`, and `dae`.

4 Creating and Referencing Objects of the User Classes

Creating and referencing an object of a user class, as required by the program that determines the index, i.e. the `daeIndexDet` program, is done as follows:

```
IDExample ex1;           // instantiates object of class Example1

IDExample * pobj = &ex1; // pobj is a pointer to the object ex1
```

The pointer `obj` is later passed on as a parameter in a function call for further calculations. See section 7 for more.

5 Parameters Controlling the Program Run

The program `daeIndetDet` works with some global parameters. The name identifiers of these parameters are defined in the header file `defaults.h`, provided in the library `indexdet.lib`. The name identifiers are listed below:

_TP_DEGREE_ The degree of Taylor coefficients or highest derivative degree.

_QR_EPS_ The threshold for the Householder QR factorization with column pivoting.

_IO_EPS_ The threshold for I/O functionalities.

_PRINT_PARAM_WHAT_ A print out parameter for controlling output. It indicates what to print out (i.e. level of detail in which the numerical results and other information from the program should be presented to the user).

_PRINT_PARAM_WHERE_ A print out parameter for controlling output. It indicates where to print out (i.e. to a data file, to the screen, or nowhere).

5.1 The degree of Taylor coefficients

The matrices needed by the matrix sequence with suitable chosen projectors used to compute the index are represented in terms of Taylor polynomials. Their degree of Taylor coefficients is defined for and used in all the Taylor arithmetic operations.

5.2 The QR and IO thresholds: Controlling the precision

5.2.1 The QR threshold

The QR threshold is the threshold for the Householder QR factorization with column pivoting. It is used when verifying whether the pivot elements hold the *epsilon* condition or not.

5.2.2 The IO threshold

The IO threshold is the threshold for I/O functionalities. For example, before printing out a matrix to the screen or to a data file it is verified whether its elements are greater or lesser than the IO threshold. In case an element is lesser than the threshold, a zero is printed out and not the digits with the whole precision.

5.3 The printout parameters: Controlling the information to be printed out

Two printout parameters can be used in combination for controlling the output of the program: `_PRINT_PARAM_WHAT_` and `_PRINT_PARAM_WHERE_`. The former indicates *what to print*, i.e., which information should be provided by the program during its execution or when it finishes. The latter indicates the desired output destiny, i.e., *where to print* the corresponding information. Both parameters have values by default predefined in the variables `_PRINT_PARAM_WHAT_VAL_` and `_PRINT_PARAM_WHERE_VAL_`, as it will be introduced in Section 6.

5.3.1 What to print

Printing out information to the user has different levels of detail, which become more and more complex according to the importance of the level. The level of the desired output is determined by the value of the variable `_PRINT_PARAM_WHAT_VAL_`, which can be controlled by the user.

Table 1 presents the different levels of detail this parameter can have.

The simplest level, i.e., when `_PRINT_PARAM_WHAT_VAL_` = 0, is recommended when the user wants to measure the running time of the program, for instance, since no other information is printed out, which might be a time consuming task.

5.3.2 Where to print

Not only what to print is important but also where to print out that information. The output destiny is determined by the value of `_PRINT_PARAM_WHERE_VAL_`, a variable that can be also controlled by the user.

Table 2 presents the different values this variable can take. Notice that defining `_PRINT_PARAM_WHERE_VAL_` = 2 implies printing no information at all, no matter the value the variable `_PRINT_PARAM_WHAT_VAL_` has (i.e., what to print out).

6 Default Settings

The global parameters are set a priori to allow the user to start using the program, no matter which values these parameters have. Table 3 shows their default

Table 1: Printout parameter: What to print.

_PRINT_PARAM_WHAT_VAL_	What to print
0	The returned code of the main program.
1	Information when _PRINT_PARAM_WHAT_VAL_ = 0. Both fatal and warning error texts, if present. Calculated index.
2	Information when _PRINT_PARAM_WHAT_VAL_ = 1. Evaluation of the functions $x(t)$, $d(x(t), t)$, and $f(z(t), x(t), t)$ at $t = t_0$. Matrices A, B, and D. Intermediate rank values. Last pivot elements holding the condition by the Householder QR. Last pivot elements not holding the condition by the Householder QR. Matrices P_0, P_0P_1, $P_0P_1P_2$, and so on. Matrices $DD^- = R$, $DP_1D^- = DP_0P_1D^-$, $DP_1P_2D^-$, and so on, as well as matrix $(DP_iD^-)'$ from each iteration.
3	Information when _PRINT_PARAM_WHAT_VAL_ = 2. Both projectors P_i and Q_i from each iteration.
22	All details.

Table 2: Printout parameter: Where to print.

_PRINT_PARAM_WHERE_VAL_	Where to print
0	to the screen
1	to a data file
2	nowhere

settings.

Table 3: Default settings for the global parameters.

Parameter name	Default value
_TP_DEGREE_VAL	10
_QR_EPS_VAL	10^{-15}
_IO_EPS_VAL	10^{-15}
_PRINT_PARAM_WHAT_VAL	0
_PRINT_PARAM_WHERE_VAL	0

6.1 Modifying the default settings

Should the default settings be changed, the user must define in the main program (i.e. in `daeIndexDet.cpp`) a variable of type `IDOptions` like this:

```
IDOptions options;
```

The variable `options` will encapsulate an object of type `IDOptions` and will provide the user with well-defined functions that update or change the default settings. Then, depending on the global parameter values she/he want to change, several alternatives are possible for defining the new values.

6.1.1 Modifying the values one by one

Making:

```
options.daeindexset( _TP_DEGREE_, 5 );
options.daeindexset( _QR_EPS_, 1E-10 );
options.daeindexset( _PRINT_PARAM_WHAT_, 2 );
```

indicates that the new value for the Taylor polynomial's degree will be 5 (i.e. `_TP_DEGREE_VAL` = 5 instead of the default value 10), that the new value for the Householder QR factorization's threshold will be 10^{-10} (i.e. `_QR_EPS_VAL` = 10^{-10} instead of 10^{-15}), and that the level for printing out information from the program will be 2 (i.e. `_PRINT_PARAM_WHAT_VAL` = 2 instead of 0).

The function `daeindexset` allows for changing the global parameters' settings. It has two parameters: the former is the name identifier of the global parameter (e.g. `_TP_DEGREE_`) and the latter is its new value (i.e., 5). By doing this, the default settings for the global parameters can be changed one by one. This overwrites the old settings.

6.1.2 Modifying several values at a time

In this case, an array of name identifiers should be declared, indicating the global parameters the user wants to change its values:

```
int nameid[] = { _TP_DEGREE_, _QR_EPS_ };
```

Then, there should be provided the new values for the desired global parameters, also in an array:

```
double value[] = { 8, 0.00001 };
```

The array `nameid` contains the name identifiers and the array `value`, their new values.

The order in which the name identifiers are typed should be the same as the order of the values in the array `value`. I.e., *_TP_DEGREE_* will have the value 8 and *_QR_EPS_* the value 0.00001. *Any unspecified parameter will have default values.*

Again, the function `daeindexset` finally does change the values:

```
options.daeindexset( 2, nameid, value );
```

where the first parameter indicates the number of global parameters that are going to be changed (two in the example), and both *nameid* and *value* are as defined above.

7 Main Function Call

The corresponding function call to the actual function that determines the index of DAEs is:

```
int errcode = daeindex( argc, argv, pobj, options );
```

or

```
int errcode = daeindex( argc, argv, pobj );
```

depending on whether the default settings were changed (first case) or not (second case).

The first function call has four parameters. Both parameters `argc` and `argv` are the well known argument vectors. They make available to a C/C++ program the arguments from its command line and its values when the program is called. The parameter `pobj` is the pointer to the object that encapsulates the user example functionalities (see Section 4). Finally, the parameter `options` is the pointer to the object that encapsulates the values of the global parameters to change (see Section 6.1).

The second function call has only three parameters, which are the same as for the first three parameters in the function call above. It works with default values for the global parameters.

The returned code of the program is controlled through the variable `errcode`.

8 All in All

Summarizing, the `daeIndexDet.cpp` program can look like follows:

```
#include "matrixseq.h"
#include "Example1.h"
#include <iostream>
#include <typeinfo>

int main( int argc, char *argv[] )
{
  int errcode;
  Example1 ex1;
  IDExample * pobj = &ex1;
  IDOptions options;

  options.daeindexset( _TP_DEGREE_, 5 );
  options.daeindexset( _QR_EPS_, 1E-10 );
  options.daeindexset( _PRINT_PARAM_WHAT_, 2 );

  errcode = daeindex( argc, argv, pobj, options );
  return errcode;
}
```

where the header `matrixseq.h` allows for using the library where the functions concerning the index determination are included; The header `Example1.h` corresponds to the class where the user defines her/his functions $x(t)$, $d(x(t),t)$, and $f(z(t),x(t),t)$, as well as all other information regarding the example to use (like the point at which the index should be computed, for instance); And the

standard C++ library headers `iostream` and `typeinfo` should be used for reading from and writing to the standard streams, and for getting information about both static and dynamic types, respectively.

9 Automatic Differentiation using ADOL-C

As it was already mentioned in Section 1, we use the C++ package ADOL-C for evaluating derivatives. This is why the vector functions related to the user problems are coded in programs written in C++ (see Section 4).

Figure 2 shows the bridge between the `daeIndexDet` program and ADOL-C. In particular, specific ADOL-C drivers are used to compute the Taylor coefficients of both the dependent and the independent variables related to the DAE, to the dynamic, and to the trajectory. On the left hand side of the figure is the user header containing the definition of the DAE, of the dynamic, and of the trajectory, together with other global parameters and input information to the program (see sections 3 and 5 for details).

The next sections will introduce the basics of the other components from the figure, specially how do the Taylor coefficients are computed. Finally, the construction of the matrices of Taylor coefficients $A(t)$, $B(t)$, and $D(t)$ (defined in Section 1.1) is also addressed.

9.1 Active Sections: Computing the Taylor coefficients

The `indexdet` library includes the header `matrixseq.h` (see Figure 1) which provides the following functions:

1. `asectra` for computing the trajectory,
2. `asecdyn` for computing the dynamic, and
3. `asecdae` for computing the DAE.

each of them including an *active section* to calculate the trajectory, the dynamic, and the DAE, respectively, as illustrated in Figure 2.

An active section in ADOL-C is a sequence of statements that contains the calculations involving the differentiable quantities at some time during the program execution. The active sections of the functions `asectra`, `asecdyn`, and `asecdae` will be described in the next sections.

9.1.1 Active section to compute the trajectory

The function `asectra` defines the variables and the active section for the trajectory, calculating it for a given t as coded by the user in the corresponding member

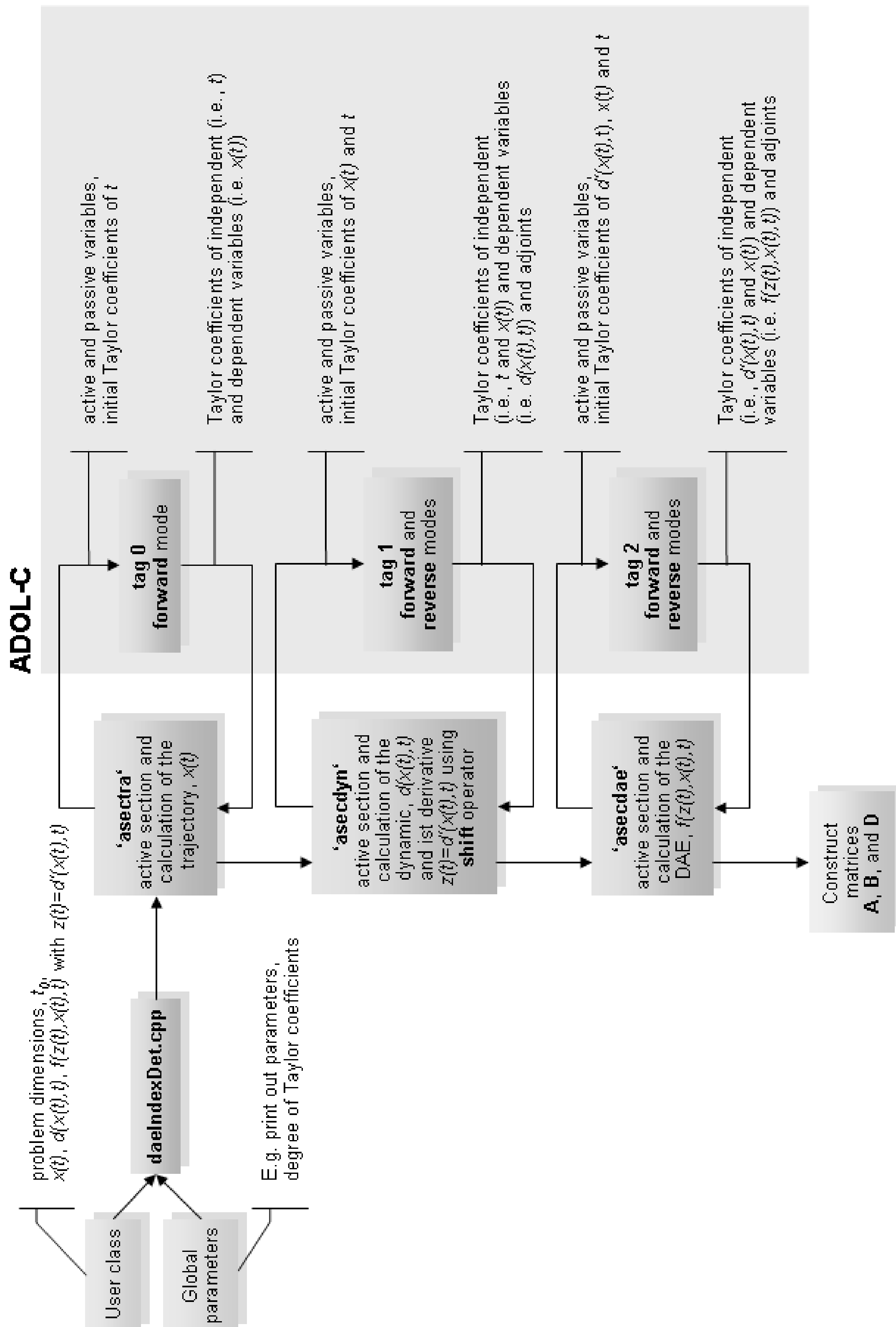

Figure 2: Automatic Differentiation using ADOL-C.

function `tra` from its polymorphic example class (e.g. the class *Example1* from Section 4).

The *tag* equal to zero distinguishes the computations related to the variables defined in this active section. The independent variable t is used to define an active variable that is later used to compute the trajectory with a call to the function `tra` via a function pointer.

The Taylor coefficients

$$X_{tra}(i) = \sum_{j=0}^{deg+1} X_{tra_j} \cdot i^j : \mathbb{R} \to \mathbb{R}^{n_{tra}} \tag{10}$$

of the independent variable t (with *deg* the degree of Taylor coefficients as introduced in Section 5.1) are initialized to the value of t given by the user in the example header class (e.g. $t_0 = 0.0$ from `Example1.h`). After this initialization, the Taylor coefficients

$$Y_{tra}(i) = \sum_{j=0}^{deg+1} Y_{tra_j} \cdot i^j : \mathbb{R} \to \mathbb{R}^{m_{tra}} \tag{11}$$

of the dependent variable $x(t)$ are computed with a call to the ADOL-C function `forward`, which implements the ADOL-C forward mode.

9.1.2 Active section to compute the dynamic

The function `asecdyn` defines the variables and the active section for the dynamic, $d(x(t), t)$, calculating it for given $x(t)$ and t as coded by the user in the corresponding member function `dyn` from its polymorphic example class (e.g. the class *Example1* from Section 4).

The independent variable t is also here an independent for the active section. The other independent active variable for the dynamic is defined using the Taylor coefficients from $x(t)$ computed by the trajectory. The dynamic is evaluated via a function pointer to the member function `dyn`, also coded by the user.

The Taylor coefficients

$$X_{dyn}(i) = \sum_{j=0}^{deg+1} X_{dyn_j} \cdot i^j : \mathbb{R}^{m_{tra}} \times \mathbb{R} \to \mathbb{R}^{n_{dyn}} \tag{12}$$

of the now independent variables $x(t)$ and t, are initialized as follows:

$$X_{dyn} = [Y_{tra}, X_{tra}] \tag{13}$$

i.e., to Y_{tra}, the matrix of Taylor coefficients from $x(t)$ (see Equation 11), and to X_{tra}, the matrix of Taylor coefficients from t (see Equation 10), already

computed in the previous section. Only the partial derivatives with respect to the first independent variable of $d(x(t), t)$, i.e., w.r.t. $x(t)$, are considered.

The Taylor coefficients

$$Y_{dyn}(i) = \sum_{j=0}^{deg+1} Y_{dyn_j} \cdot i^j : \mathbb{R}^{m_{tra}} \times \mathbb{R} \to \mathbb{R}^{m_{dyn}} \tag{14}$$

of the dependent variable $d(x(t), t)$ are computed with a call to the ADOL-C function `forward`. The Taylor coefficients Y_{dyn} can be used to compute the derivative $d'(x(t), t)$: We know from [7] that

$$\begin{aligned} p(j) &= \sum_{j=0}^{deg} p_j \cdot t^j + O(t^{deg+1}) && (15) \\ &= p_0 + p_1 \cdot t + p_2 \cdot t^2 + \ldots + p_{deg} \cdot t^{deg}. && (16) \end{aligned}$$

for a Taylor polynomial $p(t)$. Then,

$$p'(t) = p_1 + 2 \cdot p_2 \cdot t + 3 \cdot p_3 \cdot t^2 + \ldots + deg \cdot p_{deg} \cdot t^{deg-1}. \tag{17}$$

We calculate $d'(x(t), t)$ by applying a *shift* operator. In this case, the coefficients are shifted so that the new matrix $Y_{ddyn} = d'(x(t), t)$ has also $deg + 1$ columns, the last of them with zero values. These derivatives will be used in the next section, i.e., to compute the DAE.

The matrix of adjoints $Z_{dyn} \in \mathbb{R}^{m_{dyn} \times n_{dyn} \times deg}$ is used to compute the Jacobian matrix $\frac{\partial d}{\partial x}$. Computing the adjoints is done with a call to the ADOL-C function `reverse`, which implements the ADOL-C reverse mode. They will also be used in the next section.

9.1.3 Active section to compute the DAE

The function `asecdae` defines the variables and the active section for the DAE $f(z(t), x(t), t) = 0$, calculating it for given $z(t)$, $x(t)$, and t as coded by the user in the corresponding member function `dae` from its polymorphic example class (e.g. the class *Example1* from Section 4).

Again, the independent variable t is an independent for the active section. However, only the partial derivatives w.r.t. to the first and to the second independent variables of $f(z(t), x(t), t)$, i.e. w.r.t. $z(t)$ and $x(t)$, are needed. Notice that the vector of independents $y = \begin{pmatrix} z \\ x \\ t \end{pmatrix}$ is composed by $z(t)$ (i.e. the Taylor coefficients Y_{ddyn} of $d'(x(t), t)$), by $x(t)$ (i.e. the Taylor coefficients Y_{tra}), and by the Taylor coefficients from t. The DAE is evaluated via a function pointer to the member function `dae` as coded by the user.

The Taylor coefficients

$$X_{dae}(i) = \sum_{j=0}^{deg+1} X_{dae_j} \cdot i^j : \mathbb{R}^{m_{dyn}} \times \mathbb{R}^{m_{tra}} \times \mathbb{R} \to \mathbb{R}^{n_{dae}} \tag{18}$$

of the independent variable y should not be calculated anew: We have them from the matrices Y_{ddyn}, Y_{tra}, and X_{tra} (i.e. the corresponding Taylor coefficients of the independent variables computed in the sections 9.1.2 and 9.1.1, respectively). Then, the Taylor coefficients X_{dae} are initialized as follows:

$$X_{dae} = [Y_{ddyn}, Y_{tra}, X_{tra}] \tag{19}$$

The Taylor coefficients

$$Y_{dae}(i) = \sum_{j=0}^{deg+1} Y_{dae_j} \cdot i^j : \mathbb{R}^{m_{dyn}} \times \mathbb{R}^{m_{tra}} \times \mathbb{R} \to \mathbb{R}^{m_{dae}} \tag{20}$$

of the dependent variable $f(z(t), x(t), t)$ are computed with a call to the ADOL-C function `forward`.

The matrix of adjoints $Z_{dae} \in \mathbb{R}^{m_{dae} \times n_{dae} \times deg}$ is used to compute the Jacobian matrix $\frac{\partial f}{\partial y} = \left(\frac{\partial f}{\partial z}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial t}\right)$. They are computed with a call to the ADOL-C function `reverse`.

9.1.4 Constructing the matrices A, B, and D

The matrices of Taylor coefficients $A(t)$ and $B(t)$ are computed using the matrix of adjoints Z_{dae} introduced in Section 9.1.3. For the DAE coded in the header file `Example1.h` we have presented here, the Jacobian matrix $\frac{\partial f}{\partial y}$ has the form:

$$\frac{\partial f}{\partial y} = \left(\frac{\partial f}{\partial z}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial t}\right) = \left[\begin{array}{cc|ccc|c} \star_z & \star_z & \star_x & \star_x & \star_x & \star_t \\ \star_z & \star_z & \star_x & \star_x & \star_x & \star_t \\ \star_z & \star_z & \star_x & \star_x & \star_x & \star_t \end{array}\right] \tag{21}$$

each $\star$ being a pointer to a vector of deg Taylor coefficients. The first two columns correspond to the Taylor coefficients of $\frac{\partial f}{\partial z}$ (because $m_{dyn} = 2$ in the example), the next three columns correspond to the ones related to $\frac{\partial f}{\partial x}$ (since $m_{tra} = 3$), and the last column to the ones of the independent variable t. The matrices $A(t)$ and $B(t)$ have then the following form:

$$A(t) = \left(\frac{\partial f}{\partial z}\right) = \begin{bmatrix} \star_z & \star_z \\ \star_z & \star_z \\ \star_z & \star_z \end{bmatrix} \in \mathbb{R}^{m_{dae} \times m_{dyn}} \tag{22}$$

$$B(t) = \left(\frac{\partial f}{\partial x}\right) = \begin{bmatrix} \star_x & \star_x & \star_x \\ \star_x & \star_x & \star_x \\ \star_x & \star_x & \star_x \end{bmatrix} \in \mathbb{R}^{m_{dae} \times m_{tra}} \tag{23}$$

The matrix of Taylor coefficients $D(t)$ can be obtained from the matrix of adjoints Z_{dyn} introduced in Section 9.1.2. For our example it has the form:

$$D(t) = \left(\frac{\partial d}{\partial x}\right) = \begin{bmatrix} \star & \star & \star \\ \star & \star & \star \end{bmatrix} \in \mathbb{R}^{m_{dyn} \times n_{dyn}} \tag{24}$$

since $m_{dyn} = 2$ and $n_{dyn} = 3$. The partial derivatives involving the independent variable t, i.e. $\frac{\partial d}{\partial t}$, are not used.

References

[1] M. Berz et al. Computational Differentiation: Techniques, Applications, and Tools. In *Proceedings of the Second International Workshop on Computational Differentiation*, SIAM, Santa Fe, New Mexico, 1996.

[2] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in In Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 2000.

[3] A. Lamour. Index Determination and Calculation of Consistent Initial Values for DAEs. *Computers and Mathematics with Applications*, 50:1125–1140, 2005.

[4] R. März. The index of linear differential algebraic equations with properly stated leading terms. In *Result. Math.*, volume 42, pages 308–338. Birkhäuser Verlag, Basel, 2002.

[5] R. März. Differential Algebraic Systems with Properly Stated Leading Term and MNA Equations. In K. Antreich, R. Bulirsch, A. Gilg, and P. Rentrop, editors, *Modeling, Simulation and Optimization of Integrated Circuits, International Series of Numerical Mathematics*, volume 146, pages 135–151. Birkhäuser Verlag, Basel, 2003.

[6] R. März. Fine decoupling of regular differential algebraic equations. In *Result. Math.*, volume 46, pages 57–72. Birkhäuser Verlag, Basel, 2004.

[7] A. Walther, A. Kowarz, and A. Griewank. *ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++, Version 1.10.0*, July 2005.