DAETS User Guide

Version 1.1

Nedialko S. Nedialkov
McMaster University
Canada

John D. Pryce
Cranfield University
United Kingdom

Preface

Daets, Differential-Algebraic Equations by Taylor Series, is a C++ package that integrates an Initial Value Problem (IVP) for a differential-algebraic equation (DAE) system of arbitrary index and order over a range, using a Taylor series method. It can report detailed information about the structure of the DAE, as well as the numerical solution of an IVP either at the end of the range or step-by-step.

We believe the first report of solving a DAE by Taylor series, in an *ad-hoc* way, is in the work of Y.F. Chang with G.F. Corliss on the ATOMFT code c. 1994. John Pryce (JDP) learned of this on a visit to Corliss in 1996, during which, between them, they found the basic principles underlying the structural analysis approach. JDP soon developed these into a systematic method, and wrote a proof-of-concept code in Matlab in 1996, an introductory paper in 1997 and a basic theory paper [19] in 2001.

There was little progress until the collaboration with Ned Nedialkov (NSN) began after JDP acted as external examiner for NSN's PhD thesis [13] on the Vnode validated ODE solver in 1999. Vnode's architecture strongly influenced that of Daets.

We started work on theory and code of Daets in 2002 with papers [14, 15, 16]. Experience over several years with coding DAE problems, and performance tests, resulted in many improvements to the user interface and the algorithms. The first version 1.0 of the code and user guide were released in June 2008. An improved version 1.1 was released in May 2009.

We acknowledge with thanks the support given to JDP by the Leverhulme Trust and the Engineering and Physical Sciences Research Council, both of the UK, and to NSN by the Canadian Natural Sciences and Engineering Research Council. We thank the Fields Institute in Toronto, in whose stimulating environment we had the discussions that committed us to this project.

We acknowledge the help of friends and colleagues: George Corliss, for having provided the original idea and for having carefully read and constructively criticized our writings over many years, including this guide; Ole Stauning for providing assistance with his Fadbad++ automatic differentiation package, which was instrumental in the development of both Vnode and Daets; Wanhe Zhang for working on the design and many numerical experiments with an early version of Daets; Andreas Wächter for assistance with Ipopt; Andrea Walther and Andreas Griewank for helpful discussion on automatic differentiations and their Adol-C package; Rob Corless, Wayne Enright and Silvana Ilie for insightful discussions on theory, especially on methods for numerical error estimation and control; Emil Sekerinski for his input on object-oriented design; Bingzhou Zheng for extensive testing of version 1.0.

Nedialko S. Nedialkov
Department of Computing and Software
McMaster University
Hamilton, Ontario
L8S 4K1, Canada
`ned.nedialkov@reliablenumerics.com`

John D. Pryce
Department of Information Systems
Cranfield University
Shrivenham Campus
Swindon
SN6 8LA, UK
`j.d.pryce@ntlworld.com`

May 25, 2009

Contents

In the electronic version of this document, every cross-reference is a hyperlink. For instance you can click on the entry “Obtaining DAETS*” above, to jump to that section. This also applies to page numbers in the Index, and, in the body of the text, to chapter and section references and to equation numbers. To return to where you just were, use your PDF reader’s “Back” command.*

Chapter 1

Basic Usage

Daets is a C++ package for the numerical solution of initial value problems (IVPs) for differential-algebraic equation systems (DAEs). This guide assumes knowledge of basic theory of DAEs as covered for instance in the texts [2, 7]: what DAEs are; why they are in certain ways harder to solve than ordinary differential equations (ODEs); and the notion of index as a measure of a DAE's difficulty.

If you are unfamiliar with the area, but still wish to read further, consult first the brief explanation of DAEs in FAQ 5.2.12 and FAQ 5.2.13.

Section 1.1 of this chapter gives a short overview of the code's purpose and numerical methods, with references to fuller explanations later in this guide and in published work. Section 1.2 introduces the Daets user interface via a simple complete program. Section 1.3 is a reference guide to the classes and methods used in this program.

The rest of the guide is organized as follows. Chapter 2 is a reference guide to all other user-callable features. Chapter 3 gives some more substantial applications of Daets. Chapter 4 is about installation. Chapter 5 summarizes in §5.1 the theory of Daets, based on Pryce's structural analysis of DAEs, and gives answers in §5.2 to some frequently asked questions. Appendix A records changes in each new version of Daets.

1.1 Description

Daets advances the solution of a DAE system over a range using a variable-step Taylor series (TS) method. The system has the form

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

in terms of the unknown state variables $x_j(t)$, $j = 1, \ldots, n$. It is defined by a user-supplied function that evaluates the functions f_i.

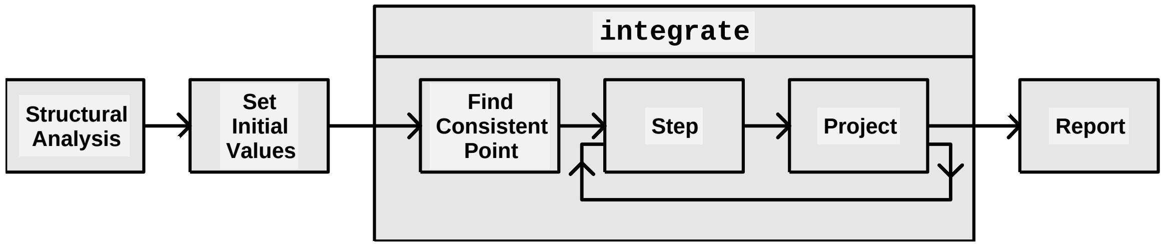
Many DAE solvers are limited by the index of the system, a non-negative integer that generally indicates how "difficult" the system is to solve numerically: see [2, 7] and Chapter 5. Daets uses a different approach that is not inherently limited by the index. It interprets the user's function code symbolically and uses automatic differentiation [6] to expand the solution in Taylor series to as many terms as required. This implies that the function code must be built of functions that are analytic along the solution path. See §1.3.1 for more details.

Unlike an ODE, a DAE usually does not have a solution through each point in the solution space, but only through initial values (IVs) that are *consistent* (§5.1). DAETS has a *structural analysis* stage (§5.1) that finds the underlying structure of the system and tells it just what variables and derivatives must be specified to define consistent IVs. This may not be obvious to the user, who must set up this data before calling the `integrate` method of DAETS. If the user gets this wrong, DAETS prints an error message, listing missing and/or superfluous IVs.

Finding a consistent initial point can be the hardest part of the solution task for highly non-linear DAEs. Many DAE solvers require exactly consistent values to be given. This is desirable but difficult, e.g. the value of a Lagrange multiplier or an acceleration is often not known *a priori*. DAETS allows guesses to be given, mixed with fixed values. For instance one may specify "The IVs are $x_1 = 2$, $x_2 = 0$ (fixed) and $x_2' = 5$, $x_3 = 0$ (guessed)". DAETS treats computation of consistent data as an optimization problem and passes this to the optimization code IPOPT [22]. During this computation, guessed values are allowed to be altered, while fixed values are left unchanged. Subject to this, the consistent point that is found is the one that is closest to the point given by the user in a least-squares sense.

The main work of DAETS is done by the `integrate` method (§1.3.2), which advances the solution, step by step, from the found consistent point `x` at the initial t=`t`, up to a final t=`tend`. Each step generates and sums a Taylor series and then does a *projection* stage to preserve consistency within the specified accuracy tolerance. Assuming one reaches `tend` successfully, one can efficiently integrate to further `tend` values as required. The TS order is chosen as a function of the tolerance, or may be set by the user, but is fixed during one call to `integrate`.

The following block diagram shows the basic order of events in solving a problem, as illustrated by the example program below.



Event location — finding a t where some function of the solution becomes zero — is not currently provided. However the integrator can be put into "one-step" mode, allowing event location to be coded in the calling program. See §3.4 for an example.

Auxiliary functions to control the integration are described in Chapter 2. For instance, the TS order may be changed, as may the tolerance and the type of accuracy control (relative, absolute, or mixed), the number of solution components printed, etc.

1.2 Quick start guide

1.2.1 An example program

A simple program is shown in Figure 1.2. It can be adapted to solve other simple problems. As given, it solves a problem with the simple pendulum system, a DAE of differential index 3, defined by the equations

$$\begin{aligned}0 &= f = x'' + x\lambda \\ 0 &= g = y'' + y\lambda - G \\ 0 &= h = x^2 + y^2 - L^2,\end{aligned} \tag{1.2}$$

see Figure 1.1. Here gravity G, and the length L of the pendulum, are constants. The independent variable is time t. The dependent (state) variables are the coordinates $x(t)$, $y(t)$ of the pendulum bob, and the Lagrange multiplier $\lambda(t)$.

The program computes the motion on $0 \le t \le 100$ with IVs $(x, y) = (-10, 0)$ and $(x', y') = (0, 1)$ at $t = 0$, taking $G = 9.8$ and $L = 10$, and displays solution values at the end of the integration.

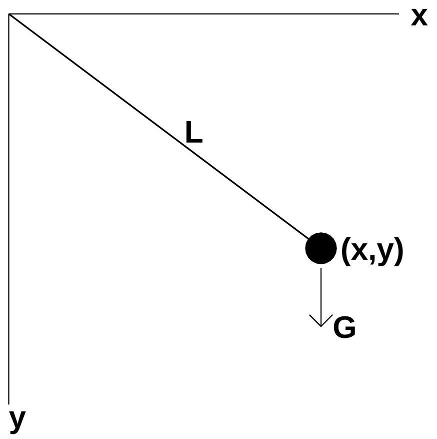


Figure 1.1: Simple pendulum.

The interface to DAETS is defined by the header file `DAEsolver.h` (code line 2). The source code of DAETS is in namespace `daets`, hence the line 3.

Lines 4–11 are the function that defines the DAE. Its name is `fcn` here, but this is arbitrary, as are the names of the arguments `t,z,f,param`. Input argument `t` represents the independent variable t. Argument `z` is an array of a templated type `T`, representing the state variable vector $\mathtt{z} = (x,\ y,\ \lambda)$. The code computes the corresponding values of the functions f, g, h in (1.2) and puts them in the array `f`, which also has type `T`. The type `T` is interpreted as several actual types at compile time—one of them is the basis for the numerical computation with Taylor series, and the others support symbolic manipulation of the `fcn` code during the preliminary structural analysis process. Arrays are indexed the C++ way, from zero, so `z[0]` means x, `f[0]` means f, etc. The operation `Diff(`$\cdot, q$`)` means $\mathrm{d}^q/\mathrm{d}t^q$, so `Diff(z[0],2)` means x'', i.e., $\mathrm{d}^2x/\mathrm{d}t^2$.

Argument `param` is explained in §1.3.1. Its use is illustrated in the Van Der Pol example of §3.3 and the continuation example of §3.4.

The main program is in lines 12–27. Lines 15–16 construct the two key objects for problem solution: `Solver`, an object of the `DAEsolver` class, which “understands” the DAE, and `x`, of the `DAEsolution` class, which may be viewed as a point moving along the solution path. `DAE_FCN` in line 15 is a macro, explained in §1.3.2.

`x` records the current values, not only of $\mathtt{z} = (x, y, \lambda)$ (the contents of `z` in `fcn`), but also of the independent variable t. Thus if, as here, t represents time, and `z` is loosely thought of as space, then `x` represents a point $(\mathtt{z}, t)$ in space-time.

Lines 17–19 give `x` its IVs. Method `setT` initializes the independent variable component, and `setX` initializes the state variable components, including their derivatives. We use the name `setX` because the state variables are called x_j in the general

```
1  // file pendulumsimple.cc in daets/examples
2  #include "DAEsolver.h"
3  using namespace daets;
4  template <typename T>
5  void fcn(T t, const T *z, T *f, void *param) {
6    // z[0], z[1], z[2] are x, y, λ.
7    const double G = 9.8, L = 10.0;
8    f[0] = Diff(z[0],2) + z[0]*z[2];
9    f[1] = Diff(z[1],2) + z[1]*z[2] - G;
10   f[2] = sqr(z[0]) + sqr(z[0]) + sqr(z[1]) - sqr(L);
11 }
12 int main() {
13   const int n = 3;                        // size of the problem
14   double t0 = 0.0, tend = 100.0;
15   DAEsolver Solver(n, DAE_FCN(fcn)); // create a solver, analyse DAE
16   DAEsolution x(Solver);                  // create a DAE solution object
17   x.setT(t0)                              // initial value of t
18    .setX(0,0,-10.0).setX(1,0, 0.0)        // .. and of x and y
19    .setX(0,1,  0.0).setX(1,1, 1.0);       // .. and of x' and y'
20   SolverExitFlag flag;
21   Solver.integrate(x, tend, flag);        // integrate until t =tend
22   Solver.printDAEtableau();
23   if (flag!=success)
24     printSolverExitFlag(flag);            // check the exit flag
25   x.printSolution();                      // display solution
26   return 0;
27 }
```

Figure 1.2: Program for simple pendulum problem.

DAE definition (1.1). The components are numbered 0, 1, . . . , as in `fcn` above, and `setX(`j`,`d`,`v`)` sets the dth derivative of the jth variable to v. Thus `x.setX(1,1, 1.0)` initializes the first derivative of the 1'th variable to 1, i.e., $y' = 1$ at $t = 0$, here.

Line 21 does the real work of solving the IVP, using the `integrate` method of the `DAEsolver` class, which moves `x` along the solution path to the desired end-point at $t = 100$. Usually, `integrate` accounts for most of the CPU time of the solution process. Since it is a numerical method, there are accuracy tolerances involved. For the default tolerance values, how to change them, and the choice between absolute, relative and mixed accuracy control, see §2.1.3.

If the integration failed to reach the end-point, this is reported by lines 23–24 using the variable `flag`, which is of an enumeration type described in §1.3.4.

The solution at $t = 100$ (or at the last t successfully reached in case of failure) is reported by the `printSolution` method (line 25). Its output is shown in Figure 1.3[1]. The dth column of the jth row of the display shows the dth derivative of the jth variable with j counting from 0 as previously. Thus row 0 shows the final values of x and x'; row 1 shows those of y and y'; while no data about λ is shown. This is because the *offsets* d_j of the variables (see §1.2.2) are 2, 2 and 0 respectively, and the behavior of `printSolution` is to display up to the $(d_j - 1)$th derivative of the jth variable for each j, in case the DAE is quasilinear, as it is here—see §1.2.2.

[1]The program also displays a few lines of "splash screen" of the IPOPT package.

```
TABLEAU
-------
        |  0   1   2  |c_i
      --------------------
       0|  2   -   0* | 0
       1|  -   2*  0  | 0
       2|  0*  0   -  | 2
      --------------------
     d_j|  2   2   0  |
        Index = 3,  DOF = 2
SOLUTION
--------
     t = 1.000000e+02
                   x                x'
     ---------------------------------------
      x0       5.683109e+00    4.563114e+00
      x1       5.950172e+00   -8.716614e+00
      x2
```

Figure 1.3: Output of the program from Figure 1.2

.

Some programming points follow.

1. The problem size `n` $= 3$ (line 13) and integration endpoints `t0` $= 0$, `tend` $= 100$ (line 14) need not be given names. For instance one could delete line 13 and replace line 15 by
   `DAEsolver Solver(3, DAE_FCN(fcn));`

2. Setting the IVs in lines 17–19 is done by a single statement containing "chained" calls to `setX`. This could be done by separate statements such as `x.setT(0.0);` followed by `x.setX(0,0, -10.0);` etc.—the choice is a matter of taste.

3. To access individual solution components, e.g., for finer control of output than offered by the convenience function `printSolution`, use `getT` and `getX`, see §2.2.

4. The IVs given in lines 17–19 happen to be consistent—the initial position $(x, y) = (-10, 0)$ exactly satisfies the "obvious" constraint $0 = h = x^2 + y^2 - L^2$, and the initial velocity $(x', y') = (0, 1)$ exactly satisfies the "hidden" constraint $0 = h' = 2xx' + 2yy'$, which means the velocity is tangential to the circle. Consistency is explained in §1.2.2 and in Chapter 5.

   When inconsistent IVs are given, DAETS does not automatically report the initial consistent point it found. If this is desired, do a first call to `integrate` with `tend` equal to `t`, which just sets `x` to a consistent point and returns. Report `x` by `printSolution`, and call `integrate` a second time to solve up to the desired `tend`.

5. DAETS has data protection features that ensure a correct set of IVs is provided (see §1.2.2), and make it impossible to inadvertently corrupt `x` in the middle of an integration (see start of §2.2).

1.2.2 Offsets, initial values, quasilinearity, consistency

In the program of Figure 1.2, how do we know—apart from physical knowledge that a pendulum requires an initial position and velocity to be given—that x, y, x' and y' are the correct set of IVs to specify? This comes from the offsets mentioned above, which are crucial to the use of DAETS. We explain them briefly here and more fully in §5.1, see also [16].

The offsets are $2n$ nonnegative integers computed in the structural analysis phase. d_j is the offset of variable x_j, and c_i is the offset of equation f_i. One way to see their role is as follows. If one differentiates the ith equation c_i times w.r.t. t, $(i = 1, \dots, n)$, one gets n equations that *in principle*, by the Implicit Function Theorem, should be solvable for the d_jth derivative of x_j $(j = 1, \dots, n)$ in terms of lower derivatives, thus converting the DAE to an ODE. This involves an $n \times n$ matrix, the *System Jacobian* $\mathbf{J}$. Broadly, the numerical method succeeds if this is found to be nonsingular.

The d_jth derivatives $x_j^{(d_j)}$ are the *leading derivatives* of the DAE. If they occur in a jointly linear way in the original (undifferentiated) f_i then the DAE is called *quasilinear*. Given the same offsets, a quasilinear problem needs fewer IVs than one that is not quasilinear. Counter-intuitively, a leading derivative need not occur anywhere in the DAE—see FAQ 5.2.6.

To recognize quasilinearity, ask "Can the DAE be written in matrix form $A\mathbf{x}+\mathbf{b} = \mathbf{0}$ where $\mathbf{x}$ is the column vector of leading derivatives, and the matrix A and vector $\mathbf{b}$ are independent of $\mathbf{x}$?"

Example. For the pendulum system (1.2), the offsets of (x, y, λ) are $(2, 2, 0)$, so the leading derivatives are x'', y'' and λ. In this example, we have

$$\begin{pmatrix} f \\ g \\ h \end{pmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x'' \\ y'' \\ \lambda \end{pmatrix} + \begin{pmatrix} 0 \\ -G \\ x^2 + y^2 - L^2 \end{pmatrix},$$

so the answer is Yes, and the DAE is quasilinear. If, say, the first equation is replaced by $0 = f = (x'')^3 + x\lambda$ or $0 = f = x''y'' + x\lambda$, the system ceases to be quasilinear.

The rule for specifying IVs can now be stated. For the reason behind it, see §5.1.

- If the DAE is quasilinear, IVs for each x_j and its derivatives up to but not including the d_jth must be given. (So a variable whose offset is zero needs no IVs.)

- Otherwise, IVs for each x_j and its derivatives through the d_jth must be given.

Example. The pendulum DAE is quasilinear, and (x, y, λ) have offsets $(2, 2, 0)$, so IVs must be given for (x, x', y, y'). If it is changed to be not quasilinear then DAETS recognizes this, and IVs must be given for $(x, x', x'', y, y', y'', \lambda)$.

One needs to know what the definition of consistent values, as follows.

- If the DAE is quasilinear, the IVs must satisfy each $f_i = 0$ and its derivatives up to but not including the c_ith. (So a function whose offset is zero imposes no constraint.)

- Otherwise, the IVs must satisfy each $f_i = 0$ and its derivatives up to and including the c_ith.

Example. For the pendulum DAE, (f, g, h) have offsets $(0, 0, 2)$ so the consistency equations for (x, x', y, y') are $0 = (h, h')$, that is $0 = x^2 + y^2 - L^2$ and $0 = 2xx' + 2yy'$. If it is changed to be not quasilinear, the enlarged set of variables $(x, x', x'', y, y', y'', \lambda)$ must satisfy the extra consistency equations $0 = f$, $0 = g$ and $0 = h'' = 2(xx'' + (x')^2 + yy'' + (y')^2$.

As a consequence, the numerical solution values held in a solution object, such as `x` in the example program, are not a flat vector as they are with an ODE. Daets uses an "irregular array" such as (for the pendulum)

$$\begin{array}{c|c|c|} & 0 & 1 \\ \hline 0 & x & x' \\ \hline 1 & y & y' \\ \hline 2 & \multicolumn{2}{l}{} \end{array} \quad \text{if quasilinear, or} \quad \begin{array}{c|c|c|c|} & 0 & 1 & 2 \\ \hline 0 & x & x' & x'' \\ \hline 1 & y & y' & y'' \\ \hline 2 & \lambda & \multicolumn{2}{l}{} \\ \cline{2-2} \end{array} \quad \text{if not.}$$

Method `setX` (§1.3.3) is used to initialize such arrays; see also `updatePoint` (§2.2.2).

1.3 Basic classes and methods

The classes, constructors and methods required for the simplest use are described here. All other methods are described in Chapter 2.

1.3.1 Defining the DAE system

Function `fcn` defines the f_i in the system (1.1). It is a *templated* function that is instantiated with several different actual types `T`, to support the automatic differentiation and structural analysis. For coding purposes, `T` can be thought of as denoting type **double**, with some restrictions described below. The specification of `fcn` is:

template <typename `T`**> void** `fcn(T t,` **const** `T *x, T *f,` **void** `*param)`

`t`

Input: denotes the variable t.

`x`

Input: Array of length n, where `x[`$j-1$`]` denotes the variable x_j for $j = 1, \dots, n$.

`f`

Output: Array of length n, where `f[`$i-1$`]` must be set to the value of f_i for $i = 1, \dots, n$.

`param`

Input, optional: pointer to a user-defined object of arbitrary type that passes parameters to the definition of the DAE.

The code of `fcn` can use the following operations on variables of type `T`.

- The arithmetic operations `+ - * /` .
- Assignment `=` and the shortcut operators `+= -= *= /=` .
- Assigning a **double** or a literal numeric constant value, as in `T sum = 0;`
- These analytic standard functions:

  `sin cos tan sqr sqrt exp pow log asin acos atan`

  (`sqr(`y`)` computes y^2).
- Differentiation (to any non-negative integer order), done by the `Diff` function. For instance, `Diff(x[j-1],2)` represents x_j'', that is, $\mathrm{d}^2 x_j/\mathrm{d}t^2$.

Notes. The code should not contain any branches, that is `if` or `switch` statements. Functions that are defined using these are not covered by the current theory.

The code may freely declare other variables. These will normally be of type **int** or **double** or `T`. Any intermediate variable that depends directly or indirectly on the inputs `x` or `t` must be of type `T`.

1.3.2 `DAEsolver` class

The constructor and the `integrate` method are described here. They are the only `DAEsolver` methods used by the program in §1.2.1. A `DAEsolver` object has many attributes describing the DAE system—the `fcn` code, the offsets, etc.—and holding strategy parameters of the integration—accuracy tolerance, Taylor series order, maximum allowed stepsize, etc. For the methods related to these see §2.1.

Constructor

```
DAEsolver(int n, T1 fcn1, T2 fcn2, T3 fcn3, T4 fcn4,
          void *param = NULL);
```

`n`
: Input: the value n, as above.

`fcn1` – `fcn4`
: Input: All four must be the same, namely the function `fcn` defining the DAE, as in §1.3.1, with the signature

  <typename `T>` **void** `fcn(T t,` **const** `T *x, T *f,` **void** `*param)`

  The repetition is due to C++ language limitations. See the Note below.

  The types `T1`, `T2`, `T3`, `T4` do not concern the user.

`param`
: Input, optional: pointer to a user-defined object of arbitrary type that passes parameters to the definition of the DAE. See §2.1.6 and §3.3 for details.

Result: This constructor does much work behind the scenes. It performs the structural analysis of the DAE and stores the offsets, and determines whether the system is quasilinear. Thus the constructed object knows the scheme for generating the Taylor series, including what set of x_j and their derivatives comprise valid IVs, and the equations they must satisfy for consistency.

Note. For convenience, the macro `DAE_FCN` is introduced to simplify a call to this constructor as, e.g.,

```
DAEsolver Solver(n, DAE_FCN(fcn));
```

and the user is recommended to call it in this way.

Ill-posed DAEs. This constructor recognizes when a DAE is ill posed; see §5.1 for a definition. If such a DAE is encoded, a solver object is still created, but it is in an “illposed” state. This can be detected by a call to the `isIllPosed` method; see §2.1.2.

`integrate` method

void `integrate(DAEsolution &x,` **double** `tend, SolverExitFlag &flag);`

`x`
: Input: contains the initial `t` and the initial values of the solution.
Output: contains the reached t and the computed values of the solution at t.

`tend`
: Input: the desired final value of t. If `tend` $<$ `t`, integration proceeds in the negative direction.
If `tend` $=$ `t`, the code uses the supplied `x` to compute a consistent initial point (if `x` is not already consistent), and returns this in `x` leaving `t` unchanged.

`flag`
: Output: reports the success of failure or the computation, see §1.3.4.

Result: This does the main work in solving the problem by advancing the integration of the DAE over a range.

1.3.3 `DAEsolution` class

A `DAEsolution` object `x` holds a “point” comprising the current values of the independent variable t, and of the x_j and their derivatives at t, as described in §1.2.2; for more details see Chapter 5. It also stores various data related to the state of the integration.

Constructor

`DAEsolution(`**const** `DAEsolver &Solver)` **throw**`(std::logic_error);`

`Solver`
: Input: a `DAEsolver` object.

Result: A `DAEsolution` object associated with the given `Solver`. It is in the `Initial` state, see start of §2.2. Before it can be used by the `integrate` method, it must be given initial values by means of its `setT` and `setX` methods.

If `Solver` is constructed from an ill-posed DAE, an exception `std::logic_error` results.

`setT` method

```
DAEsolution & setT(double t) throw(std::logic_error);
```

`t`
: Input: scalar. `x.setT(t)` sets $t = $ `t` in a `DAEsolution` object.

Returns: a reference to the updated `DAEsolution` object.

Constraint: `x` must be in the `Initial` state, see start of §2.2. If `x` is not in this state, an exception `std::logic_error` results.

`setX` method

```
DAEsolution & setX(int index, int order, double value,
        VarType type=Free) throw(std::logic_error,std::out_of_range);
```

`index, order, value`
: Input: such that `x.setX(` $j-1, k, v$ `)` sets the entry of `x` representing $x_j^{(k)}$, the kth derivative of the jth variable, equal to v.

Constraint: (j, k) must be in the index set J of (2.5). Otherwise, an exception `std::out_of_range` results.

`type`
: Input: optional, default `Free`; specifies, only for the initial computation of a consistent point, whether the value in question must be kept unchanged (`Fixed`) or may be altered (`Free`); see also §2.2.

Constraint: There are limits on how many IVs can be made `Fixed`. This is explained in §1.3.5.

Returns: A reference to the updated object.

Constraint: `x` must be in the `Initial` state, see start of §2.2. If `x` is not in this state, an exception `std::logic_error` results.

Notes. Returning a reference lets one "chain" calls, as in

```
x.setT(...).setX(...)...
```

For instance, to specify "The initial t is 1 and the IVs are $x_1 = 2$ (fixed) and $x_2 = 0$, $x_2' = 5$ (guessed)", the calling program could do

```
x.setT(1.0).setX(0,0, 2.0,Fixed).setX(1,0, 0.0).setX(1,1, 5.0);
```

The following is equivalent, but displays the setting of each x_j separately:

```
x.setT(1.0);
```

```
    x.setX(0,0, 2.0,Fixed);
    x.setX(1,0, 0.0);
    x.setX(1,1, 5.0);
```

A `DAEsolution` object knows which initial values it requires. If a value is set for an unnecessary (`index`,`order`) pair, `setX` displays an error message to this effect. If a required value is left unset, `integrate` detects this when called and displays an error message.

WARNING: If a value for the third parameter, `value`, is not given, the corresponding $x_j^{(k)}$ may not be set correctly. For example, in

```
    x.setX(0,0,Free);
```

x_0 is set to 2. The reason is that `Free` is an enumerated constant with value 2 (see the definition of `VarType` in §2.2), C++ converts enumerated values to `int`, and this call sets x_0 to 2.

`printSolution` **method**

void `printSolution(ostream &s=cout)` **const**;

`s`
: Input: output stream, default is `cout`.

`x.printSolution(s)` prints the t and x components into the output stream `s`.

t is displayed first. Then for $j = 1, \dots, n$ on separate lines, labeled 0 to $n-1$ to match C++ indexing, the derivatives $x_j^{(k)}$ are displayed. k runs from 0 to d_j-1 if the DAE is quasilinear, and from 0 to d_j if not.

1.3.4 `SolverExitFlag` type

This enumerated data type provides an error indicator for the `integrate` method of `DAEsolver`.

Definition

typedef enum `SolverExitFlag`
`{success, toofewdof, nonconsistentpt, uninitializedpt, htoosmall};`

`success`	The integration has completed successfully.
`toofewdof`	More IVs were specified "fixed" than the number of degrees of freedom of the DAE.
`nonconsistentpt`	DAETS was unable to find an initial consistent point.
`uninitializedpt`	`setT` is not called on the input `x` of `integrate`, or `setX` is not called for all the required components of `x`.
`htoosmall`	The stepsize during integration became too small to make progress.

`printSolverExitFlag` **method**

void `printSolverExitFlag(SolverExitFlag state, ostream &s = cout);`

`state`
Input: Usually, the value returned by a call to the `integrate` method.

`s`
Input: output stream, default is `cout`.

Result: `printSolverExitFlag` outputs into `s` a message describing the meaning of `state`.

1.3.5 Fixed and free initial values

Clearly, there must be limits on how many IVs one can specify as "fixed". For instance in the pendulum system, one cannot give random values to both x and y and fix both of them, because the constraint $x^2 + y^2 = L^2$ generally will not be satisfied.

The rules for what one can fix are entirely determined by the offsets. One has to explain at this point that the irregular array $\mathbf{X}$ of values that define a consistent point, see (2.5), divides into disjoint parts that are computed in stages, and have a natural indexing by "stage numbers" k in the range $-d \le k \le \alpha$, where $d = \max_j d_j$, and α is -1 if the DAE is quasilinear, 0 otherwise.

Namely, at stage k, to have consistent values, one must solve the equations

$$f_i^{(k)} = 0 \qquad \text{for those } i = 1, \ldots, n \text{ for which } k + c_i \ge 0$$

for the unknowns

$$x_j^{(k)} \qquad \text{for those } j = 1, \ldots, n \text{ for which } k + d_j \ge 0.$$

We illustrate with the pendulum, assuming it has been changed to be non-quasilinear. The figure below shows the situation. Irregular array $\mathbf{X}$ consists of $(x, y, x', y', x'', y'', \lambda)$.

$k =$	-2	-1	0	1	$\dots$
d_j	Unknowns				
2	x	x'	x''	x'''	$\dots$
2	y	y'	y''	y'''	$\dots$
0			λ	λ'	$\dots$
c_i	Equations				
0			$f = 0$	$f' = 0$	$\dots$
0			$g = 0$	$g' = 0$	$\dots$
2	$h = 0$	$h' = 0$	$h'' = 0$	$h''' = 0$	$\dots$
Fixable	$2 - 1 = 1$	$2 - 1 = 1$	$3 - 3 = 0$	$\dots$	

(The figure also shows the reason for the term “offset”—each list of derivatives is shifted left by the relevant offset, relative to the k=0 column.) The process starts at stage $k = -2$ because the largest d_j is 2. At stage $k = -2$, one must find x and y that satisfy $h(x, y) = 0$. This is one equation for two unknowns, so there is $2 - 1 = 1$ degree of freedom (DOF) at this stage, which is how many of x and y one is allowed to specify as fixed (the “Fixable” row in the table).

At stage $k = -1$, one must find x' and y' that satisfy $h'(x', y') = 0$. (Actually h' involves x and y also, but these have been solved for, and are no longer unknowns.) Again there is one equation for two unknowns, so again $2 - 1 = 1$ DOF, which is how many of x' and y' one is allowed to specify as fixed.

At stage $k = 0$, one has three equations for three unknowns (x, y, x', y' also occur in the equations, but are no longer unknowns), so there are no DOF, and one cannot specify any of the unknowns x'', y'', λ of this stage as fixed. In fact it is clear, whether the DAE is quasilinear or not, that stage $k = 0$ always has n equations for n unknowns, so none of its unknowns can be fixed.

The general rule is that only stages $k < 0$ are relevant to “fixing”: the number of equations at stage k is $m_k =$ (the number of i for which $k + c_i \geq 0$); the number of unknowns at stage k is $n_k =$ (the number of j for which $k + d_j \geq 0$); and the number of unknowns that can be fixed at stage k ($-d \leq k < 0$) is the difference $n_k - m_k$.

Consider the pendulum example, and suppose its length $L = 10$. One can fix x, or y, or neither, but not both. If one fixes x to be 8, say, there are two roots for y, namely $y = -6$ and $y = 6$. Hence an appropriate guess for y should be given, and Ipopt should then find the root that is nearest it. Clearly, one can not necessarily impose an arbitrary fixed value: if we fix x at a value > 10, then no solutions exist.

If one fixes neither x nor y, then Ipopt will find an approximation to the consistent point nearest to the given guess of (x, y). For instance if guesses $(x, y) = (4, -3)$ are given, the found consistent values will be close to $(8, -6)$.

This happens similarly on each stage $k < 0$.

Summary

This chapter has described enough of the DAETS methods, and of their underlying theory, to let you write a simple program like that in Figure 1.2. The next chapter documents all the remaining methods, organized by the three DAETS classes—`DAEsolver`, `DAEsolution` and `DAEpoint`—that are visible to the user. If you are more interested in examples of code using DAETS, see Chapter 3. For installation matters, see Chapter 4.

Chapter 2

All Other Methods

Sections 2.1, 2.2 and 2.3 of this chapter describe the remaining methods of `DAEsolver` and `DAEsolution`, and the methods of `DAEpoint`, which is a base class of `DAEsolution`. If a method is not found here, it is in §1.3. The final Section 2.4 gives a list of default parameter values, for reference.

There are frequent forward references to Chapter 5. They tell you where to go *if* you want to look up an item of structural analysis theory. You can understand the gist of this chapter without doing so.

2.1 Controlling integration: the `DAEsolver` class

The method descriptions in this section assume a `DAEsolver` object, call it `Solver`, has been constructed from a DAE function called `fcn`, and that the DAE's functions and variables are f_i and x_j respectively, as in equation (1.1).

Structural analysis, touched on in §1.2.2 and detailed in §5.1, is done during execution of the `DAEsolver` constructor. Subsections 2.1.1 and 2.1.2 describe methods that get results of this analysis or display them on standard output.

During integration the stepsize h changes from step to step. The Taylor series (TS) order p is fixed over one call to `integrate`. Either it is chosen by the code on entry to `integrate`, as a function of the tolerance, or the user program sets it before calling `integrate`. Subsections 2.1.3 onward outline aspects of the integration algorithm, alongside the methods by which the user can tune its behavior by changing "strategy parameters" to do with h, p and the accuracy tolerance.

These parameters are attributes of a `DAEsolver` object, call it `Solver`. If one wants to solve the same problem using, say, different Taylor orders p, one may either use one `Solver` and change its p between integrations, or use several `Solver`s, each having a p that does not change. The choice is a matter of convenience.

2.1.1 Access to structural analysis data

The first three methods use the `vector` class of the C++ standard library.

`getSigmaMatrix` method

```
void getSigmaMatrix(vector< vector<int> > & s, int neginfval=-1)
                       throw(std::logic_error);
```

`s`
: Output: A matrix, which must be $n \times n$. If not, an exception `std::logic_error` results.

`neginfval`
: Input: Optional, the value that encodes $-\infty$. -1, because **int** does not have an "infinity" value and the alternative of using a large negative number looks ugly in printed output.

  If `neginfval` ≥ 0 an exception `std::logic_error` results.

Result: Stores in `s` the signature matrix $\Sigma = (\sigma_{ij})$ of the DAE, defined in equation (5.1).

`getCVector` method

```
void getCVector(vector<int> & c) const;
```

`c`
: Output: A vector, which must be of length n. If not, an exception `std::logic_error` results.

Result: Stores in `c` the vector of offsets c_i of the functions f_i.

`getDVector` method

```
void getDVector(vector<int> & d) const;
```

`d`
: Output: A vector, which must be of length n. If not, an exception `std::logic_error` results.

Result: Stores in `d` the vector of offsets d_j of the variables x_j.

`getNumDegsOfFreedom` method

int `getNumDegsOfFreedom()` **const**;

Returns: The number of degrees of freedom of the DAE, given by equation (5.3).

`getStructuralIndex` method

int `getStructuralIndex()` **const**;

Returns: The structural (or Taylor) index of the DAE, given by equation (5.6).

`isIllPosed` method

bool `isIllPosed()` **const**;

Returns: **true** if `fcn` describes an ill-posed DAE (see §5.1), else **false**.

`isQuasilinear` method

bool `isQuasilinear()` **const**;

Returns: **true** if the DAE is deemed to be quasilinear (see §1.2.2), else **false**.

2.1.2 Reporting structural analysis data

`printDAEinfo` method

void `printDAEinfo(ostream &s=cout)` **const**;

`s`
Input: output stream, default is `cout`.

Result: Outputs into `s`

- whether the problem is quasilinear;
- size of DAE;
- index of DAE.

For the pendulum example, this method prints:

```
DAE
---
     quasilinear
     size.................3
     index................3
```

printDAEtableau method

void `printDAEtableau(ostream &s=cout)` **const**;

`s`

Input: output stream, default is `cout`.

Result: Outputs into `s`

– signature matrix of the DAE indicating the positions of a highest-value transversal (HVT);

– offsets of the problem;

– index;

– number of degrees of freedom (DOF).

For definitions of these items, see Chapter 5.

For the pendulum example, this method prints:

```
TABLEAU
-------

         |  0   1   2  |c_i
       --------------------
        0|  2   -   0* | 0
        1|  -   2*  0  | 0
        2|  0*  0   -  | 2
       --------------------
      d_j|  2   2   0  |

          Index = 3,  DOF = 2
```

`printDAEpointStructure` **method**

void `printDAEpointStructure(ostream &s = cout)` **const**;

`s`
: Input: output stream, default is `cout`.

Result: Outputs into `s`

- the structure of a solution point, indicating which variables and derivatives of them need to be initialized.

For the pendulum example, this method prints:

```
POINT STRUCTURE
---------------
     variable   derivatives
        x0        0   1
        x1        0   1
        x2        -
```

2.1.3 Accuracy control

There is an accuracy tolerance, `tol`, and a type of accuracy control, `esttype`. At each step in the numerical solution, an estimate $\mathtt{est}_j^{(k)}$ of the local error in $x_j^{(k)}$ is computed internally, for each (j, k) in the set J as in (2.5). For the current step to be accepted, the following condition must be satisfied:

$$r \le 1, \quad \text{where} \quad r = \max_{(j,k)\in J} \frac{\left|\mathtt{est}_j^{(k)}\right|}{\tau_r \times \left|x_j^{(k)}\right| + \tau_a}, \tag{2.1}$$

and τ_r and τ_a are defined by the table:

esttype	τ_r	τ_a	Description
Abs	0.0	tol	absolute accuracy control
Rel	tol	ϵ	relative accuracy control
AbsRel	tol	tol	mixed accuracy control

.

Here ϵ is a modest multiple of the machine precision. Error theory for TS methods is less developed for DAEs than for ODEs, but there are good reasons to expect this to give a robust accuracy control.

If (2.1) is not satisfied, the stepsize is reduced, and the solution is recomputed on the current step. If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then `esttype` should be set to `Abs` on entry, whereas if the error requirement is in terms of the number of correct significant digits, then `esttype` should be set to `Rel`. For a mixed accuracy test, `esttype` should be set to `AbsRel`. The default value `AbsRel` should be chosen unless there are good reasons for a different choice.

`setTol` method

This method lets the program alter the accuracy tolerance and the type of accuracy control from the default of a "mixed" test with tolerance 10^{-8}.

Calling `setTol();` with no arguments resets the working values to their defaults.

```
void setTol(double tol=1e-8, ErrorEstType esttype=AbsRel)
          throw(std::logic_error);
```

`tol`

Input: the accuracy tolerance; default 10^{-8}.
Constraint: `tol` $\in [10^{-16}, 10^{-1}]$. If `tol` is outside this range, an exception `std::logic_error` results.

`esttype`

Input: The type of accuracy control; default is a mixture of absolute and relative accuracy control. `esttype` must be one of the three values of the enumeration type:

```
typedef enum ErrorEstType {Abs, Rel, AbsRel};
```

`getTol` method

```
double getTol() const;
```

Returns: the current tolerance.

`getRTol` method

```
double getRTol() const;
```

Returns: the current τ_r in (2.1).

`getATol` method

```
double getATol() const;
```

Returns: the current τ_a in (2.1).

`getErrorEstType` method

```
ErrorEstType getErrorEstType() const;
```

Returns: the current type of accuracy control.

2.1.4 Controlling the Taylor series order

We denote the order `integrate` uses by p. The Taylor series of the jth variable x_j is computed up to the term of order $p + d_j$, where d_j is the offset of x_j. The minimum and maximum allowed orders are denoted by $p_{\min}$ and $p_{\max}$ respectively.

Currently, $p_{\min} = 1$. The maximum number of Taylor coefficients that can be generated is set to `MaxLength` $= 80$. Then the largest allowed value of p, for a given DAE, is

$$p_{\max} = \texttt{MaxLength} - 1 - \max_j d_j. \tag{2.2}$$

When the constructor of `DAEsolver` is executed, it checks that $p_{\max} \geq p_{\min}$. If this is not the case, an exception `std::out_of_range` is thrown with a message stating the smallest value for `MaxLength` that has to be given when DAETS is compiled[1].

When `integrate` is allowed to select the order itself (because `setOrder`, below, is not called, or is called with argument 0) it uses a heuristic formula that has usually performed well:

$$p = \max\big\{p_{\min},\ \min\{\lceil -0.5 \ln(\texttt{tol}) + 1 \rceil,\ p_{\max}\}\big\}, \tag{2.3}$$

Here `tol` is the tolerance given to DAETS, and $\lceil x \rceil$ is the next integer $\geq x$. In the absence of the limits $p_{\min}$ and $p_{\max}$, and with the allowed range of tolerances (see `setTol` method), this gives values in the range 2 to 18.

Tests on many problems have shown that the CPU time to solve a problem, as a function of p for fixed `tol`, usually has a flat minimum, with the best order p_{best} in the range 15 to 20, and slowly increasing as `tol` is decreased. Further, it is *much* safer to take p too large rather than too small. A small p like 1 or 2 can increase solution time by orders of magnitude, while taking $p = 2 \times p_{\text{best}}$, say, typically increases solution time quite modestly.

[1] If you would like `MaxLength` increased, contact N. Nedialkov who can produce a precompiled library with larger `MaxLength`.

`setOrder` **method**

```
void setOrder(int order) throw(std::out_of_range);
```

`order`
Input: the order to which Taylor series are generated.
Constraint: $p_{\min} \le$ `order` $\le p_{\max}$ or `order` $= 0$.

- If `order`$= 0$, `integrate` selects the order by the heuristic (2.3).
- If `order` is in $[\,p_{\min}, p_{\max}\,]$, `integrate` uses order $p =$ `order`.
- If `order` does not satisfy the above constraint, exception `std::out_of_range` is thrown with a message saying that the order must be in $[\,p_{\min}, p_{\max}\,]$ or 0.

`getOrder` **method**

```
int getOrder() const;
```

Returns: the current order, p.

`getMinOrder` **method**

```
int getMinOrder() const;
```

Returns: $p_{\min}$.

`getMaxOrder` **method**

```
int getMaxOrder() const;
```

Returns: $p_{\max}$.

2.1.5 Controlling the stepsize

On each integration step after the first, `integrate` computes a trial stepsize h for the next step, based on the current h and the ratio r in (2.1), such that a prediction of the estimated error is below a tolerance. The stepsize is restricted by the values of the largest, $h_{\max}$, and smallest, $h_{\min}$, allowed stepsizes.

If the trial h satisfies $|h| < h_{\min}$, this is a fatal error, and the integration is terminated. If $|h| > h_{\max}$, h is replaced by $\pm h_{\max}$, with the same sign as h.

By default, $h_{\max}$ is set to the largest finite double precision number, and can be set to a smaller value by the user. The usual reason for doing so is that the solution path is observed to have long smooth stretches during which h builds up, alternating with short regions of rapid change that may be missed altogether if h is too large. The continuation problem in §3.4 is an example.

Let $\tau = \max\{|t_0|, |t_{\text{end}}|\}$, where the integration is from t_0 to t_{end}. Then

$$h_{\min} = 16 \cdot (\tau_+ - \tau), \tag{2.4}$$

where τ_+ is the next double precision number after τ. $h_{\min}$ cannot be changed by the user.

`getHmin` method

double `getHmin()` **const**;

Returns: current $h_{\min}$.

`getHmax` method

double `getHmax()` **const**;

Returns: current $h_{\max}$.

`setHmax` method

void `setHmax(`**double** `hmax);`

`hmax`
Input: scalar.

Result: sets $h_{\max}$ to `hmax`.

If the computed by `integrate` $h_{\min} > h_{\max}$, an exception `std::logic_error` occurs. In this case, $h_{\max}$ should be set to at least $h_{\min}$.

2.1.6 Passing data to a DAE function

`passDataToDAE` method

```
void passDataToDAE(void *param);
```

`param`
: Input: pointer to data that need to be passed to the function `fcn` that specifies the DAE, through the `void *param` parameter in `fcn`.

  If `param` is the `NULL` pointer, a call to `passDataToDAE` has no effect. The user should ensure that data is correctly extracted inside `fcn`.

  For examples illustrating the use of this function see §3.3 and §3.4.

2.2 Solution path properties: the `DAEsolution` class

In the method descriptions of this section, `x` denotes a `DAEsolution` object at the current point t on the solution path $x(t)$, and $\mathbf{X}$ denotes the values of x_j and derivatives that `x` holds. That is, $\mathbf{X}$ is the irregular vector comprising $X_{jk} = x_j^{(k)}$, the computed value at t of the kth derivative of the variable $x_j(t)$, for (j, k) in the index set

$$J = \{(j,k) \mid j = 1, \ldots, n,\ 0 \le k \le d_j + \alpha\}, \tag{2.5}$$

where $\alpha = -1$ if the DAE is quasilinear, $\alpha = 0$ if not—see §1.2.2.

The basic role of an object `x` of the `DAEsolution` class is to store the numerical values t and $\mathbf{X}$ that jointly describe the current point on the solution path. However, `x` stores other important attributes.

It records its position along the solution path using the enumerated type:

> **typedef enum** `SolutionState`
> {`Initial, InitialConsistent, OnPath, EndOfPath`};

A newly created `x` is in state `Initial`, and DAETS deems it an inconsistent point. Its t, and each component of its $\mathbf{X}$, is flagged "uninitialized", and `integrate` will not accept `x` until all these values are set. When consistent values at the initial t_0 are found, `x` moves into state `InitialConsistent`; this can be seen if integrate is called with $t_{\text{end}} = t_0$ to find a consistent point and then exit. After the first accepted step, `x` is in state `OnPath` and is deemed consistent. It remains in this state until (a) a call to `integrate` fails (with a `htoosmall` or `nonconsistentpt` error), putting `x` into the `EndOfPath` state, or (b) it is returned to the `Initial` state by a call to `setFirstEntry`, see §2.2.3.

Only in the `Initial` state can the value of $(t, \mathbf{X})$ be set by the `setT` and `setX` methods, or by the `updatePoint` method of §2.2.2. In all other states, trying to alter the t or $\mathbf{X}$ values in `x` results in an exception `std::logic_error`.

`x` also records for each element of $\mathbf{X}$ whether it has been given an initial value, and (for the algorithm that finds a consistent point) whether this value was set "fixed" or "free" by the relevant `setX` call. This is recorded through the *type* of an element of `x`, declared as

typedef enum `VarType {Uninitialized, Fixed, Free};`

(Hence, using `setX`, one can also mark an entry in `x` as uninitialized.) Calling `integrate` when **X** is not fully initialized causes an `uninitializedpt` error.

`x` also has a "one-stepping" attribute, set by `setOneStepMode`, see §2.2.3. When this is true, `x` is in one-step mode, and `integrate` returns after each successful step with `x`. On the *initial* step there is a special feature: when returning on the first call, `x` contains a consistent point, but t is not changed; when returning on the second call, `x` contains a solution at the next selected t. This simplifies writing output to a file—see §3.3 for an example.

At present, one-step mode is the only way to program event location, or to produce output at each step for plotting, etc. However, a related feature is provided by `setPrintProgress` of §2.2.4, which puts `x` into a mode where the current t, step size, etc., are displayed at each step with an optional pause to help one read them. This can help debug an integration that seems to get "stuck", without the need to program a one-step loop in the main program.

Apart from the operations in this section, `DAEsolution` objects support the arithmetic operations of their base class `DAEpoint`, which just holds an irregular array **X**. Such operations "strip out" the non-`DAEpoint` attributes. For instance let `x1` and `x2` be two computed solutions at t_{end} of the same DAE. We wish to find how different they are by comparing their **X** arrays. To do so, form the difference `x1-x2`, which is a `DAEpoint` object, and then `(x1-x2).norm()`, which returns the max-norm of the difference,

$$(\texttt{x1} - \texttt{x2}).\texttt{norm}() \;=\; \max_{(j,k)\in J} \left| x_{1,j}^{(k)} - x_{2,j}^{(k)} \right|,$$

in an obvious notation, where J is the index set in (2.5).

`x` records statistics such as the number of accepted and rejected steps used so far in the integration. It also holds the attribute that determines whether the integration is done in one-step mode.

2.2.1 Access

getT method

double `getT()` **const throw**`(std::logic_error);`

Returns: `x.getT()` returns the current t stored in `x`.

Constraint: t must be initialized by a call to `setT`. Otherwise an exception `std::logic_error` results.

`getType` method

```
VarType getType(int index, int order) const throw(std::out_of_range);
```

`index, order`
: Input: such that `x.getType(` $j{-}1,k$ `)` returns the type of the entry of `x` representing $x_j^{(k)}$, the kth derivative of the jth variable.

  Constraint: (j,k) must be in the index set J of (2.5). Otherwise, an exception `std::out_of_range` results.

Returns: `x.getType(` $j{-}1,k$ `)` returns the type of the entry of `x` representing $x_j^{(k)}$.

`getX` method

```
double getX (int index, int order) const
                              throw(std::logic_error, std::out_of_range)
```

`index, order`
: Input: values j and k such that `x.getX(` $j{-}1,k$ `)` returns the current value of the entry of `x` representing $x_j^{(k)}$, the kth derivative of the jth variable.

  Constraint: If `x` is in `Initial` state, (j,k) must be in the index set J of (2.5). Otherwise, `index` $\in \{0,\dots,n-1\}$ (i.e., $j \in \{1,\dots,n\}$) and $k \in \{0,\dots,p+d_j\}$, where p is the order of the Taylor series.

Returns: `x.getX(` $j{-}1,k$ `)` returns the current value of the entry of `x` representing $x_j^{(k)}$.

An exception `std::logic_erorr` results if

- `x` is in an `Initial` state, and entry $(j-1,k)$ is not initialized by a call to `setX` or
- `x` is in an `Initial` state, and $(j-1,k)$ is not in the index set J. For example, if `x` has not been integrated, this exception arises when attempting to access an entry with indices "outside" J.

An exception `std::out_of_range` results if variable index $j \notin \{1,\dots,n\}$ or derivative index $k \notin \{0,\dots,p+d_j\}$.

2.2.2 Copying

No assignment operator `y = x` is provided for this class because copying the entire contents of `x` to `y` could compromise the locking security feature if either object is not in the `Initial` state. However, one often wants to use the **X** contents of an object to initialize another object. The `updatePoint` method offers this, without bypassing the security feature.

`updatePoint` method

`DAEsolution & updatePoint(`**`const`**` DAEpoint &x) `**`throw`**`(std::logic_error);`

`x`
Input: a `DAEpoint` or a `DAEsolution` object.

Returns: `y.updatePoint(x)` has the effect of copying the **X** part of `x` into that of `y`.

If `x` is a `DAEpoint` object:

- if the type of a component of `y` is `Uninitialized`, it becomes `Free`;
- otherwise, the type of a component of `y` remains unchanged.

If `x` is a `DAEsolution` object, the type of each element is also copied.

Constraints: `y` must be in the `Initial` state, else an exception `std::logic_error` results.

`x` and `y` must have the same shape (have the same index set J in (2.5)), else an exception `std::logic_error` results. It is not necessary that `x` and `y` were constructed from the same `Solver`.

This feature is convenient for copying the elements of initialized objects. For example, if an integration proves problematic near some $t = t_1$, and one wishes to experiment with the accuracy control or other parameter settings around this point, one can do

```
DAEsolution x(Solver);
/* initialize x
... integrate x from t0 up to some t1 */
DAEsolution y(Solver);
/* copy the values stored in x, and their types */
y.updatePoint(x);
/* ... set t and other values in y, and integrate onward from t1 */
```

2.2.3 State control

The rationale of these methods is discussed at the start of §2.2.

`setFirstEntry` method

```
void setFirstEntry();
```

Result: `x.setFirstEntry()` puts `x` into the `Initial` state so it can be (re)-initialized.

It also resets to zero the variables that gather the statistics of §2.2.5.

`setOneStepMode` method

```
void setOneStepMode(bool onestep);
```

`onestep`
: Input: if `onestep==true`, `integrate` returns after it accepts a step.

If `onestep==false`, `integrate` attempts to reach `tend` from current $(t, \mathbf{X})$ stored in `x`.

2.2.4 Reporting during integration

`setPrintProgress` method

The `setPrintProgress` method switches a `DAEsolution` object `x` in and out of a `printprogress` mode where the progress of integration is displayed. If `x` is in this mode then `integrate` prints, at each accepted step,

> the current t, number of steps (as given by `getNumAccSteps` in §2.2.5), stepsize, and local error estimate,

in a fixed position on the screen without scrolling.

```
void setPrintProgress(double s=0);
```

`s`
: Input: If `s` ≥ 0, `printprogress` mode is switched on. In addition, the integration pauses for `s` seconds after each output, to make it easier to read what is displayed.

If `s` < 0, `printprogress` mode is switched off.

Note. The operation of this method may be sensitive to your terminal settings. Please notify the authors in case of problems.

`setOutputFunction` method

This method sets a user-defined function, say `outfcn`, for data output inside the `integrate` method, for instance to write solution values to a file for plotting. For an example, see §3.6. The `outfcn` function is called at the end of each integration step.

Such a function should be declared as

```
void outfcn(const DAEsolution &x, void *out_param, void *problem_param)
```

`x`
: Input: reference to a solution object.

`out_param`
: Input: pointer to a user-defined object of arbitrary type that passes output-related parameters (e.g. file handles) to this function.

`problem_param`
: Input: pointer to a user-defined object of arbitrary type that passes problem-related parameters to this function.

To make `integrate` use `outfcn`, call on a `DAEsolution` object

```
void setOutputFunction(OutputFcn outfcn, void *out_param,
                                         void *problem_param=NULL)
```

`outfcn`
: Input: pointer to an output function.

`out_param`, `problem_param`
: Input: pointers to user-defined objects of arbitrary type; see above.

Here `OutputFcn` is defined as

```
typedef void (*OutputFcn)(const DAEsolution &, void *, void *);
```

2.2.5 Statistics

The following statistics are cumulative, starting at zero for an object `x` in the `Initial` state, and added to by each subsequent call to `integrate()` on `x` unless reset to zero by a call to `setFirstEntry`.

`getCPUtime` method

```
double getCPUtime() const;
```

Returns: user CPU time in seconds used to integrate a `DAEsolution` object. This is obtained by a call to the Fortran `etime` utility, which accesses the high-resolution clock and is typically accurate to a microsecond or better.

`getNumAccSteps` method

int `getNumAccSteps()` **const**;

Returns: number of accepted steps.

`getNumRejSteps` method

int `getNumRejSteps()` **const**;

Returns: number of rejected steps.

2.2.6 Printing

`printSolutionState` method

void `printSolutionState(ostream &s = cout)` **const**;

`s`
: Input: output stream, default is `cout`.

Result: `x.printSolutionState(s)` outputs into `s` the current state of `x` along the solution path, which is one of the values of `SolutionState`.

`printStats` method

void `printStats(ostream &s=cout)` **const**;

`s`
: Input: output stream, default is `cout`.

Result: `x.printStats(s)` outputs into `s` the following data related to the integration of `x`. Where relevant these are cumulative, as in §2.2.5.

- user CPU time
- total number of steps, number of accepted and number of rejected steps
- values of smallest and largest stepsizes
- current order of the Taylor series
- current tolerances τ_r and τ_a, see equation (2.1).

2.3 Arithmetic on solutions: the `DAEpoint` class

The `DAEpoint` class is a base class for `DAEsolution`. A `DAEpoint` object stores an irregular array, i.e., values representing derivatives $x_j^{(k)}$ for (j, k) in an index set J, see (2.5).

2.3.1 Constructors

```
DAEpoint(const DAEsolver &Solver) throw(std::logic_error);
```

`Solver`
Input: a `DAEsolver` object.

Result: Constructs a `DAEpoint` object with the correct shape for `Solver`. If `Solver` is constructed from an ill-posed DAE, an exception `std::logic_error` results.

```
DAEpoint(const DAEpoint &X);
```

`X`
Input: a `DAEpoint` or a `DAEsolution` object.

Result: Constructs a `DAEpoint` object with the same shape as `X` and copies the contents of `X` into it.

2.3.2 Access

`setX` method

```
DAEpoint & setX(int index, int order, double value)
               throw(std::out_of_range);
```

`index, order, value`
Input: such that `x.setX(` $j-1, k, v$ `)` sets the entry of `x` representing $x_j^{(k)}$, the kth derivative of the jth variable, equal to v. This is explained in §1.3.5.

Constraint: (j, k) must be in the index set J of (2.5). Otherwise, an exception `std::out_of_range` results.

Returns: A reference to the updated object.

Returning a reference lets one “chain” calls as with the `setX` method of `DAEsolution`; see §1.3.3.

getX method

double getX(**int** index, **int** order) **const**
    throw(std::out_of_range);

index, order
: Input: such that x.getX($j{-}1, k$) returns the current value of the entry of x representing $x_j^{(k)}$, the kth derivative of the jth variable.

  Constraint: (j, k) must be in the index set J of (2.5). Otherwise, an exception std::out_of_range results.

Returns: x.getX($j{-}1, k$) returns the current value of the entry of x representing $x_j^{(k)}$.

getNumVariables method

int getNumVariables() **const**;

Returns: n, number of variables a DAEpoint object stores.

getNumDerivatives method

int getNumDerivatives(**int** j) **const** **throw**(std::out_of_range);

j
: Input: variable number
  Constraint: $0 \leq$ j $\leq n-1$. If j < 0 or j $> n-1$, exception std::out_of_range is thrown, which also prints the range for j.

Returns: $d_j + 1 + \alpha$, number of derivatives a DAEpoint object stores for variable j; cf. (2.5).

Note. When getNumDerivatives(j) returns 0, it implies that variable x_{j+1} cannot be accessed by setX and getX methods. In such a case, for example, X.setX(j,0,value) would result in std::out_of_range exception.

2.3.3 Assignment

`operator =` **method**

`DAEpoint &` **operator** `= (`**const** `DAEpoint & X);`

`X`

Input: a `DAEpoint` or a `DAEsolution` object.

Result: copies the content of `X` to the object on the left in `=`, e.g. `Y=X`.

Returns: reference to the updated `DAEpoint` object.

`DAEpoint &` **operator** `= (`**double** `a);`

`a`

Input: a `double` scalar.

Result: `X=a` sets all entries of `X` equal to `a`.

Returns: reference to the updated `DAEpoint` object.

2.3.4 Arithmetic

In the operators that follow, the arguments must have the same shape, that is the irregular arrays $\mathbf{X}$ that they hold must be indexed over the same J in (2.5). If not, an exception `std::logic_error` is thrown, which also prints the reason for the "mismatch". The symbol • stands for any of `+`, `-`, `*`, and `/`.

For convenience in the notation below, by `X`(i,j) we denote the value for $x_i^{(j)}$ stored in object `X`.

`operator •=` **method**

```
DAEpoint & operator •= (const DAEpoint &X)
                    throw(std::logic_error);
```

`X`

Input: a `DAEpoint` or a `DAEsolution` object.

Returns: Reference to `*this` such that $(*$`this`$)(j,k)$ •= `X`(j,k) for all $(j,k) \in J$.

`operator •` **method**

```
DAEpoint operator • (const DAEpoint &X, const DAEpoint &Y)
                    throw(std::logic_error);
```

`X`, `Y`
: Input: `DAEpoint` or `DAEsolution` objects.

Returns: A `DAEpoint` object containing `X`(j,k) • `Y`(i,j) for all $(i,j) \in J$.

2.3.5 Comparison

`operator ==` **method**

```
bool operator == (const DAEpoint &X, const DAEpoint &Y)
                  throw(std::logic_error);
```

`X`, `Y`
: Input: `DAEpoint` or `DAEsolution` objects.

Returns: **true** if `X(i,j) == Y(i,j)` for all $(i,j) \in J$, **false** otherwise.

`operator !=` **method**

```
bool operator != (const DAEpoint &X, const DAEpoint &Y)
                  throw(std::logic_error);
```

Returns: `X != Y` is equivalent to `!(X == Y)`.

2.3.6 Norm

`norm` **method**

```
double norm() const;
```

Returns: `X.norm()` returns $\max_{(j,k)\in J} |X_{jk}|$ where $\mathbf{X}$ is the irregular array stored in `X`.

2.4 DAETS default settings

The table below lists the default values of all parameters that control the operation of the integration process, or are otherwise relevant to the user.

tolerance	10^{-8}
accuracy control	mixture of relative and absolute
order	selected automatically by DAETS according to (2.3), or set by the user
min order	1
max length of TS	80; changeable by recompiling the source
max order	determined by (2.2)
stepsize	variable, determined by DAETS
min stepsize	determined by DAETS according to (2.4)
max stepsize	the largest finite double precision number

Summary

All the user-callable features of DAETS have been covered in this chapter and Chapter 1. Many, but by no means all, of them are illustrated in the example programs of the next chapter.

Chapter 3

Examples of DAETS Code

We begin in §3.1 with suggestions for how to handle a DAE whose structure is unfamiliar to you. In §3.2 we show code to solve a small DAE from the Test Set [12], illustrating a technique to simplify the translation from Fortran, where arrays are usually indexed from 1, into C++ which indexes them from 0. In §3.3 we solve Van Der Pol's ODE, showing how parameters are passed to a DAE function and how one-step mode may be coded. §3.4 shows how DAETS can solve continuation problems. In §3.5 we give code to solve a model of putting a golf ball, originally approximated in [1] by an ODE. We show a more realistic formulation as an index 3 DAE, though the problem is really a boundary value problem and our main program does not handle this aspect. In §3.6, we illustrate how an output function is set, so `integrate` outputs data after each accepted step.

All the code shown in this chapter is included in the DAETS distribution.

3.1 Handling an unfamiliar DAE

If a DAE is new to you, you probably do not know the offsets and other results of its structural analysis, nor the index set J in (2.5), i.e. for what j and k must $x_j^{(k)}$ be initialized. Even when this is known, it may be unclear what consistent initial values look like.

To remedy this, you might write first a main program that merely creates a `Solver` and calls the reporting functions in §2.1.2. The output, particularly of `printDAEpoint-Structure()`, will give the needed structural information. Note `printDAEtableau()` displays an $n \times n$ matrix, so it should be used with caution when n is large.

For the `chemakzo` program of §3.2 below, instead of lines 58 onwards one can initially just write

```
x.printDAEinfo();
x.printDAEtableau();
x.printDAEpointStructure();
return 0;
}
```

which produces:

```
DAE
---

     not quasilinear
     size.................6
     index................1

TABLEAU
-------

        |  0   1   2   3   4   5  |c_i
      --------------------------------
       0|  1*  0   0   0   0   -  | 0
       1|  0   1*  -   0   -   0  | 0
       2|  0   0   1*  0   0   -  | 0
       3|  0   -   0   1*  0   -  | 0
       4|  0   0   0   0   1*  0  | 0
       5|  0   -   -   0   -   0* | 0
      --------------------------------
     d_j|  1   1   1   1   1   0  |

        Index = 1,  DOF = 5

POINT STRUCTURE
---------------
     variable   derivatives
        x0        0   1
        x1        0   1
        x2        0   1
```

With the output from this, you know which components to initialize. Suppose, as in this problem, you know consistent initial values, or reasonable guesses of them, for some required components—here, x_j for $j = 0$ to 4 (counting from 0 as in the main program). The program should set these values suitably, and set the others—here, x_5 and x'_0 to x'_4—to an arbitrary value such as zero. (See `setInitialValues()` at line 73 in the code below.)

If you want to inspect the consistent point that results, call `integrate` with t_{end} equal to the initial t_0. This computes a consistent point and exits. The result here is

```
SOLUTION
--------
     t = 0.000000e+00
                  x               x'
     -------------------------------------
      x0       4.440000e-01   -5.097682e-02
      x1       1.230000e-03   -1.372932e-02
      x2       0.000000e+00    2.548743e-02
      x3       7.000000e-03   -3.916080e-06
```

If on inspection this appears to be correct, you can continue the integration to the desired t_{end}. The results are shown in §3.2 below.

3.2 Chemical Akzo Nobel problem

This IVP is a stiff non-linear DAE of size 6 and index 1. It describes a chemical process, in which 2 species are mixed, while carbon dioxide is continuously added, to produce a product. It is taken from the Test Set for IVP Solvers [12], which describes more fully its formulation and origin. We give a basic program to integrate this problem and check the accuracy of the solution by computing the number of significant correct digits as defined in [12].

A useful coding technique is shown. The Test Set supplies definitions of the DAEs in Fortran, with arrays indexed from 1. Using `#define` statements, we convert between this style and C++'s style of indexing from 0. Hence, the Fortran code can be adapted with very little change.

```
// file chemakzo.cc in daets/examples
// A program to integrate the Chemical Akzo Nobel DAE problem
// from the Test Set for IVP solvers.
#include "DAEsolver.h"
using namespace daets;
/*--------------------- DAE DEFINITION -----------------*/
// Function for evaluating f(y) in formulation of Akzo Nobel DAE
template <typename T>
void fcn0(T t, const T *x, T *f) {
  // Since we adapt the Fortran code given in this test set, it is
  // convenient to introduce these two macros:
  #define f(i)      f[i-1]
  #define y(i)      x[i-1]
  // constants from problem definition
  double
    k1   = 18.7e0,
    k2   = 0.58e0,
    k3   = 0.09e0,
    k4   = 0.42e0,
    kbig= 34.4e0,
    kla = 3.3e0,
    ks   = 115.83e0,
    po2 = 0.9e0,
    hen = 737e0;
  // intermediate variables
  T r1   = k1*(pow(y(1),4))*sqrt(y(2)),
    r2   = k2*y(3)*y(4),
    r3   = k2/kbig*y(1)*y(5),
    r4   = k3*y(1)*(sqr(y(4))),
    r5   = k4*(sqr(y(6)))*sqrt(y(2)),
    fin = kla*(po2/hen-y(2));
  // the f(y) functions
  f(1) =   -2e0*r1 +r2 -r3      -r4;
  f(2) = -0.5e0*r1              -r4      -0.5e0*r5 + fin;
  f(3) =         r1 -r2 +r3;
  f(4) =            -r2 +r3 -2e0*r4;
  f(5) =             r2 -r3          +r5;
  f(6) = ks*y(1)*y(4)-y(6);
  #undef f
  #undef y
```

```
}
// fcn evaluates the Akzo Nobel DAE functions.
template <typename T>
void fcn(T t, const T *y, T *f, void *param) {
  fcn0(t, y, f);
  for ( int i = 0; i < 5; i++ )
    f[i] -= Diff(y[i], 1);
}
/*-------------- FORWARD FUNCTION DECLARATIONS------------*/
double compSignificantCorrectDigits(DAEsolution &x);
void setInitialValues(DAEsolution &x);
/*---------------------- MAIN PROGRAM --------------------*/
int main() {
  int n = 6;
  double tend = 180.0;
  // Create solver and solution point, and set initial values.
  DAEsolver Solver(n, DAE_FCN(fcn));
  DAEsolution x(Solver);
  setInitialValues(x);
  // Integrate.
  SolverExitFlag flag;
  Solver.integrate(x, tend, flag);
  if (flag!=success) {printSolverExitFlag(flag); return 1;}
  // Print solution and statistics about the integration.
  x.printSolution();
  x.printStats();
  // How did it compare with reference solution?
  printf(" *** Significant correct digits: %5.1f\n",
         compSignificantCorrectDigits(x));
  return 0;
}
/*------------------ SET INITIAL VALUES ------------------*/
void setInitialValues(DAEsolution &x) {
  double t = 0;
  x.setX(0, 0, 0.444)   .setX(0, 1, 0)
   .setX(1, 0, 0.00123).setX(1, 1, 0)
   .setX(2, 0, 0.0)     .setX(2, 1, 0)
   .setX(3, 0, 0.007)   .setX(3, 1, 0)
   .setX(4, 0, 0.0)     .setX(4, 1, 0)
   .setX(5, 0, 0)
   .setT(t);
}
/*------------------- CHECK ACCURACY ----------------------*/
// Check solution against reference solution from test set.
double compSignificantCorrectDigits(DAEsolution &x) {
  double y[6];
  // reference solution from the test set
  y[0] = 0.1150794920661702e0;
  y[1] = 0.1203831471567715e-2;
  y[2] = 0.1611562887407974e0;
  y[3] = 0.3656156421249283e-3;
  y[4] = 0.1708010885264404e-1;
  y[5] = 0.4873531310307455e-2;
  // Compute max norm of componentwise relative errors.
  double error_norm = 0;
  for (int i=0; i < 6; i++) {
    double r = fabs((x.getX(i,0)-y[i])/y[i]);
    if (r>error_norm) error_norm = r;
  }
  return -log10(error_norm);// # significant correct digits
}
```

The output is

```
    SOLUTION
    --------
         t = 1.800000e+02

                     x                x'
         --------------------------------------
          x0       1.150795e-01   -2.265538e-04
          x1       1.203831e-03    1.358514e-07
          x2       1.611563e-01    1.127593e-04
          x3       3.656156e-04   -1.036671e-06
          x4       1.708011e-02    1.380017e-06
          x5       4.873531e-03

    STATISTICS
    ----------
         TIME..............0.201          ORDER................11
         STEPS...............133          TOLERANCE
              accepted.......132               relative...1.0e-08
              rejected.........1               absolute...1.0e-08
                  %.........0.8
         STEPSIZES
              smallest.....0.048
              largest .....1.752
 *** Significant correct digits:   8.5
```

3.3 One-step mode. Passing data to a problem

We illustrate how parameters are passed to the function for evaluating the DAE in DAETS. Here, we integrate the Van Der Pol's equation

$$0 = -x'' + \mu(1 - x^2)x' - x \tag{3.1}$$

with $(x, x') = (3, 2)$ at $t_0 = 0$, values for parameter $\mu = 1, 10,$ and 100, and $t_{\text{end}} = 20, 60,$ and 500, respectively. We also show how the one-step mode is called. We report the solution on each step into files and plot x versus t and x' versus x.

We list and explain the DAETS program for integrating this problem.

```
// file vdp.cc in daets/examples
// It solves Van Der Pol's ODE. It illustrates how parameters can be
// passed to the DAE and how the one-step mode is called.
#include "DAEsolver.h"
using namespace daets;
/*------------------------- DAE DEFINITION ---------------*/
```

```
 7 template <typename T>
 8 void fcn(T t, const T *x, T *f, void *param) {
 9   // Convert the content at param to double.
10   double mu = *((double*)param);
11   // $0 = -x'' + \mu(1-x^2)x' - x$
12   f[0] = -Diff(x[0],2)+mu*(1-sqr(x[0]))*Diff(x[0],1)-x[0];
13 }
14 /*------------------------- MAIN PROGRAM -----------------*/
15 int main() {
16   double t0 = 0;
17   double mu[]   = {1,  1e1, 1e2};
18   double tend[] = {20, 60,  500};
19   double hmax[] = {0.05,0.005,0.001};
20   // Create a solver and pass the address of mu[0] as last parameter.
21   DAEsolver Solver(1, DAE_FCN(fcn), mu);
22   // Create DAEpoint object and store initial point.
23   DAEpoint xp(Solver);
24   xp.setX(0,0,3).setX(0,1,2);
25   // Create DAEsolution object
26   DAEsolution x(Solver);
27   // Set one-step mode.
28   x.setOneStepMode(true);
29   // Monitor the integration.
30   x.setPrintProgress();
31   SolverExitFlag flag=success;
32   for (int i=0; i<3; i++)
33     {
34       printf("Integrating VDP with mu=%.1e  tend = %.1f\n",
35              mu[i], tend[i]);
36       if (i>0) {
37           Solver.passDataToDAE(&mu[i]);
38           x.setFirstEntry();
39       }
40       x.updatePoint(xp).setT(t0);
41       // Create file name and open a file for writing.
42       char buf[20];
43       sprintf(buf,"vdp-%.1e.out", mu[i]);
44       FILE *fd = fopen(buf, "w");
45       // Restrict the stepsize, so the plots appear "smooth".
46       Solver.setHmax(hmax[i]);
47       // Integrate and write $t$, $x$, $x'$ at each step into file vdp-mu.out.
48       while (x.getT()!=tend[i] && flag==success) {
49         Solver.integrate(x, tend[i], flag);
50         fprintf(fd, "%.3e %.6e  %.6e\n",
51                 x.getT(), x.getX(0,0), x.getX(0,1));
52       }
53       fclose(fd);
54     }
55   if (flag != success) {printSolverExitFlag(flag); return 1;}
56   return 0;
57 }
```

When passing parameter(s) to `fcn`, the `DAEsolver` constructor must be invoked with a third parameter that is the address of a variable (line 21) that will be passed to `fcn`. Since this parameter is of type **void** `*`, the user must extract the needed value(s) inside `fcn`, line 10. Here, inside `fcn`, the data at `param` is converted to **double**, the type used in this function.

If integrations with different parameter values are desired, these values should be passed by calling the `passDataToDAE` method, line 37. Here, we pass in a loop the values 10 and 100 for μ.

WARNING:

(a) The `DAEsolver` constructor and `passDataToDAE` must be called with the same type of their corresponding parameters.

(b) The size n of a problem is fixed at the time a `DAEsolver` object `Solver` is constructed. You can not use `passDataToDAE` to change n for an existing `Solver`. If you attempt this, a segmentation fault or undefined behavior is likely to occur.

An `fcn` can allow different values of n, as in the continuation example of §3.4, but to use several values of n in one program run, you must create a different solver for each n.

We store the initial values (x, x') in a `DAEpoint` object, lines 23 and 24. Then we create a `DAEsolution` object and activate the one-step mode in line 28. In the **for** loop, before `x` is reinitialized, `setFirstEntry` must be called, line 38, to unlock `x` for further use. Since DAETS does not provide continuous output yet, which can be used for producing "smooth" plots, we restrict the largest stepsize the solver takes, line 46. (The values in `hmax` are found by trial and error.)

The plots in Figure 3.3 of x versus t and x' versus x, for $\mu = 1, 10,$ and 100, show the behavior for which this ODE is famous. From the initial point, the path moves towards a limit cycle to which it rapidly converges, and which shows abrupt changes evidenced by the "corners" near $(\pm 2, 0)$. The larger is μ, the sharper are these corners.

The GNUPLOT file used to produce these plots is

```
# file vdp.gp in daets/examples/gnuplot
# This file plots x vs t and x' vs x
# for the Van Der Pol's equation
set terminal postscript enh color eps  dl 2 "Courier" 28
### mu = 1
set title '{/Symbol m} = 1' font "Times,28"
set output 'vdp0x.eps'
set xlabel "t"  font "28"
set ylabel "y"  font "28"
plot 'vdp-1.0e+00.out' u 1:2 notitle with  lines lw 3
set output 'vdp0xxp.eps'
set xlabel "y"  font "28"
set ylabel "y'" font "28"
plot 'vdp-1.0e+00.out' u 2:3 notitle with  lines lw 3
### mu = 10
set title '{/Symbol m} = 10' font "Times,28"
set output 'vdp1x.eps'
set xlabel "t"  font "28"
set ylabel "y" font "28"
```

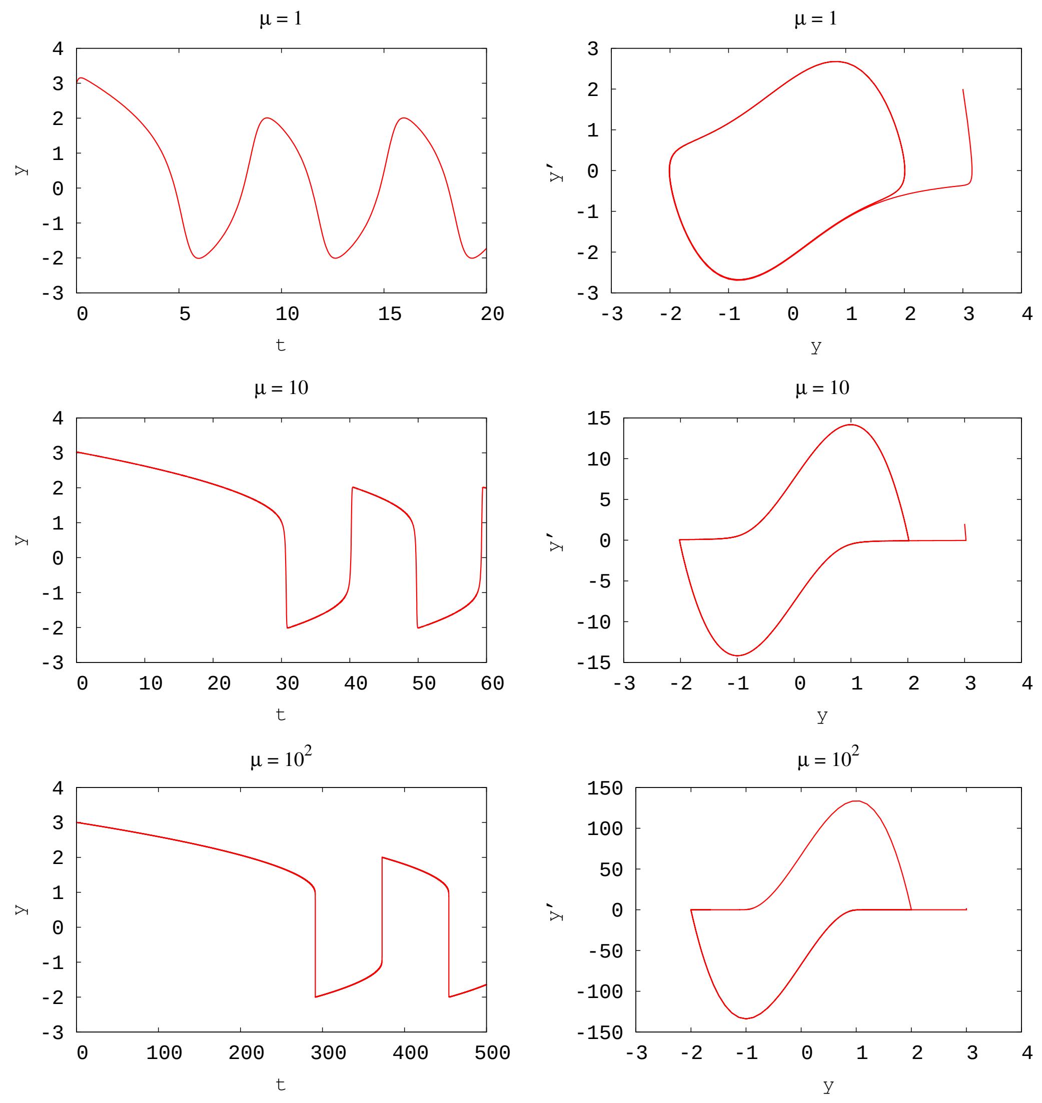


Figure 3.1: Plots of $x(t)$ versus t and $x'(t)$ versus $x(t)$ for the Van der Pol's equation.

```
plot 'vdp-1.0e+01.out' u 1:2 notitle with  lines lw 3
set output 'vdp1xxp.eps'
set xlabel "y" font "28"
set ylabel "y'" font "28"
plot 'vdp-1.0e+01.out' u 2:3 notitle with  lines lw 3
### mu = 100
set title '{/Symbol m} = 10^{2}' font "Times,28"
set output 'vdp2x.eps'
set xlabel "t"  font "28"
set ylabel "y"  font "28"
plot 'vdp-1.0e+02.out' u 1:2 notitle with  lines lw 3
set output 'vdp2xxp.eps'
set xlabel "y"  font "28"
set ylabel "y'" font "28"
plot 'vdp-1.0e+02.out' u 2:3 notitle with  lines lw 3
```

3.4 A continuation problem

This problem comes from Layne Watson [23]. We seek a fixed point, that is a root of $\mathbf{x} = \mathbf{g}(\mathbf{x})$, for the nonlinear function $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$ defined by

$$g_i(\mathbf{x}) = g_i(x_1, \ldots, x_n) = \exp(\cos(i \cdot \sum_{k=1}^{n} x_k)), \quad i = 1, \ldots, n. \tag{3.2}$$

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

$$\mathbf{f}(\lambda, \mathbf{x}) = \mathbf{0}, \tag{3.3}$$

where $\mathbf{f}(\lambda, \mathbf{x}) = \mathbf{x} - \lambda\mathbf{g}(\mathbf{x})$. When $\lambda = 0$, equation (3.3) has the trivial solution $\mathbf{x} = \mathbf{0}$, and we hope to track a solution all the way to the desired root at $\lambda = 1$.

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

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

For general problems of the form (3.3), this approach works as long as $\mathbf{f}_\mathbf{x} = \partial\mathbf{f}/\partial\mathbf{x}$ is nonsingular, but typically it has a serious weakness: one reaches values of λ where $\mathbf{f}_\mathbf{x}$ becomes singular, while the $n \times (n+1)$ matrix $[\mathbf{f}_\mathbf{x},\ \mathbf{f}_\lambda]$ retains full rank n. These are *turning points* with respect to λ.

A powerful alternative is to treat all of $x_1, \ldots, x_n, \lambda$ as on the same footing, instead of viewing λ as special, and to use *arc-length continuation*. The system (3.3) may be rewritten as n equations in $n+1$ unknowns:

$$\mathbf{f}(\mathbf{y}) = \mathbf{0}, \tag{3.4}$$

where $\mathbf{y} = (\mathbf{x}; \lambda)$. "Generically", $\mathbf{f}_\mathbf{y}$ is always of full row rank, and there is a (unique except for direction) solution path through any point satisfying (3.4), tangential to the 1-dimensional null space of $\mathbf{f}_\mathbf{y}$. Invent a new independent variable s and add to (3.4) an equation $\|\mathrm{d}\mathbf{y}/\mathrm{d}s\|_2^2 = 1$, that is

$$0 = S = \sum_j {y'_j}^2 - 1, \tag{3.5}$$

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

This formulation gives an index 1 system of $n+1$ equations and variables. To match array indexing in C++ we label these from 0 to n with λ being variable 0 and (3.5) being equation 0. The offsets are $c_0 = 0$, $c_1, \ldots, c_n = 1$ and all $d_0, \ldots, d_n = 1$. The $(n+1) \times (n+1)$ System Jacobian has $2(\mathbf{y}')^T$ as its first row, with $\mathbf{f}_\mathbf{y}$ beneath, and is nonsingular if $\mathbf{f}_\mathbf{y}$ has full rank, since $\mathbf{y}'$ is orthogonal to the rows of $\mathbf{f}_\mathbf{y}$.

How does DAETS know which direction to take along the path? The code recognizes the resulting DAE as not quasilinear and thus requires values up to stage $k = 0$ as an initial guess. Since all offsets d_j are 1, these values are $y_1, \ldots, y_{n+1}$; $y'_1, \ldots, y'_{n+1}$ — not only an initial position, but also an initial direction. This is just what is needed.

Code to solve the Layne Watson problem by arc-length continuation is given below (at a smaller font size because of its length). The main program uses two loops: to output solution values in one-step mode, and to locate $\lambda = 1$ by a Newton iteration.

```
// file laynewatson.cc in daets/examples
#include <cmath>
#include "DAEsolver.h"
using namespace daets;
/*---------------------- DAE DEFINITION ----------------*/
template <typename T>
void compDAE(T s, const T *x, T *f, void *param ) {
  // The param argument conveys the problem size:
  int nplus1 =  *((int*)param);
  int n = nplus1-1;
  T lambda = x[0];
  // Define f[0] = function S that specifies s to be arc length:
  f[0] = -1.0;
  for (int k=0; k<nplus1; k++)
    f[0] += sqr(Diff(x[k],1));
  T sum = 0.0;
  for (int k=1; k<=n; k++)
    sum += x[k];
  // The algebraic equations are f[1] to f[n]
  for (int i=1; i<=n; i++){
    f[i] = x[i] - lambda * exp(cos(i*sum));
  }
}
/*-----------------FORWARD FUNCTION DEFINITION------------*/
static void outputfcn(FILE *fd, const DAEsolution &x, int nplus1);
/*------------------------- MAIN PROGRAM -----------------*/
int main() {
  int n = 10, nplus1 = n+1;
  DAEsolver Solver(nplus1, DAE_FCN(compDAE), &nplus1);
  DAEsolution x(Solver);
  double t0  = 0.0, tend;
  int order  = 15;
  double tol = 1e-7;
  // Initial guess has x and x' all 0
  //   except x(0)', that is d(lambda)/ds, is +1,
  //   which hopefully gets us going in right direction
  x.setX(0, 0,  0.0).setX(0, 1, 1.0);
  for (int i=1; i<=n; i++)
    x.setX(i, 0,  0.0).setX(i, 1, 0.0);
  x.setT(t0);
  Solver.setOrder(order);
  Solver.setTol(tol);
  x.setOneStepMode(true);
  FILE *fd = fopen("laynewatson2.out", "w");
  SolverExitFlag flag = success;
  while ( x.getX(0,0)<1 && flag == success ) {
    // recall x(0,0) is lambda
    Solver.integrate(x, x.getT()+1, flag);
    outputfcn(fd, x, nplus1);
  }
  printSolverExitFlag(flag);
  x.printStats();
  // Now we know lambda>=1 at this step and was <1 at previous step.
  // DAETS doesn't do event location yet, so hand-code this here.
  x.setOneStepMode(false);
  while ( fabs(x.getX(0,0)-1.0 ) > tol && flag == success ) {
    x.setFirstEntry();
    tend = x.getT() - (x.getX(0,0)-1.0)/x.getX(0,1);
    Solver.integrate(x, tend, flag);
    outputfcn(fd, x, nplus1);
  }
  printSolverExitFlag(flag);
  x.printStats();
  x.printSolution();
  if (flag!=success) return 1;
  return 0;
}
/*----------------FUNCTION FOR OUTPUT TO FILE-------------*/
// This function outputs into a file suitable for GNUPLOT
static void outputfcn(FILE *fd, const DAEsolution &x, int nplus1)
{
  fprintf(fd, "%.4e ", x.getT() );
  for (int i=0; i<nplus1; i++) {
    fprintf(fd, "   " );
    for (int k=0; k<2; k++)
      fprintf(fd, "%.7e ", x.getX(i,k) );
  }
  fprintf(fd, "\n");
}
```

The output of this program to the terminal is

```
EXIT STATE
----------
     SUCCESS: The integration has completed successfuly

STATISTICS
----------
     TIME..............3.387          ORDER................15
     STEPS.............513            TOLERANCE
          accepted.......509               relative...1.0e-07
          rejected.........4               absolute...1.0e-07
              %.........0.8
     STEPSIZES
          smallest..3.563e-04
          largest .....0.667

EXIT STATE
----------
     SUCCESS: The integration has completed successfuly

STATISTICS
----------
     TIME..............0.016          ORDER................15
     STEPS.................1          TOLERANCE
          accepted.........1               relative...1.0e-07
          rejected.........0               absolute...1.0e-07
              %.........0.0
     STEPSIZES
          smallest.....0.629
          largest .....0.629

SOLUTION
--------
     t = 8.750393e+01

                 x               x'
     --------------------------------------
      x0      1.000000e+00    6.442725e-02
      x1      1.491914e+00    1.691229e-01
      x2      5.066654e-01    7.231676e-02
      x3      3.890434e-01    4.516114e-03
      x4      9.273171e-01   -1.377331e-01
      x5      2.419807e+00   -1.464676e-01
      x6      2.186966e+00    5.771084e-01
      x7      7.729182e-01    3.289186e-01
      x8      3.720929e-01    4.789042e-02
```

```
x9        5.865923e-01    -2.006262e-01
x10       1.753840e+00    -6.616551e-01
```

From the program's output to a file, the plot in Figure 3.2 was produced.

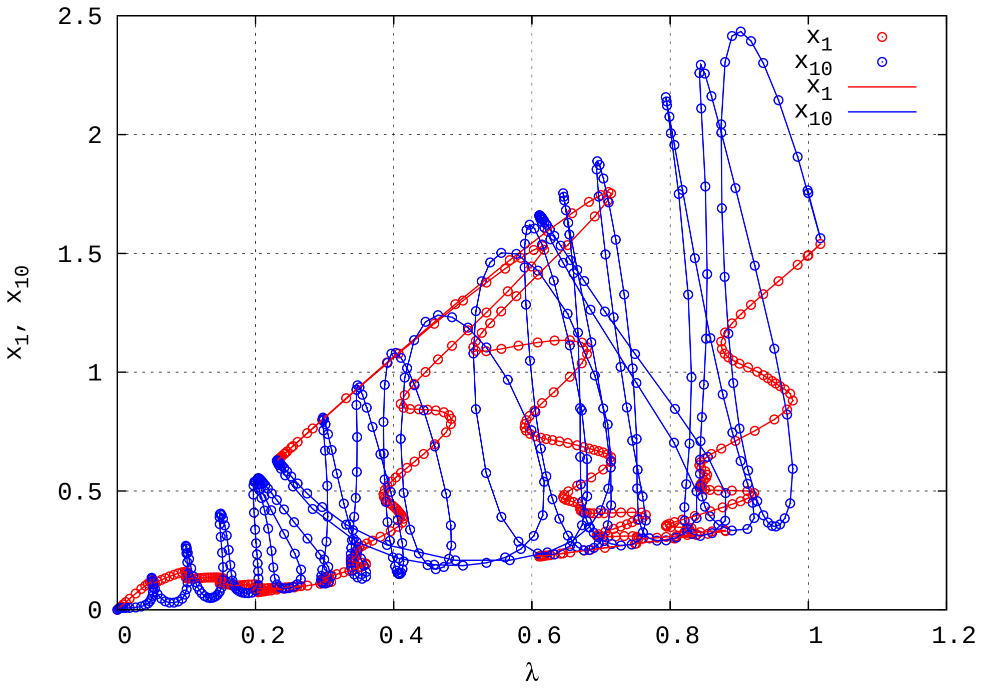


Figure 3.2: Layne Watson problem, $n = 10$. The paths of x_1 and x_{10} are plotted against the original parameter λ. Many turning points can be seen. The markers show successive integration points. Points beyond $\lambda = 1$ are part of the event location.

Tests showed that for reliable results one should impose a stepsize limit by the `setHmax()` method (not done in this code). Otherwise, one may jump from the correct path to another very close one. A limit of 0.3 worked for all experiments we tried, namely every value of n from 2 to 40, and several larger n. With many combinations of order and tolerance, we always found the same solution at $\lambda = 1$. Many others exist (one finds some of them by continuing the path beyond the first one found), but "the first occurrence of $\lambda = 1$ on the path starting from $\lambda = 0$" specifies a unique solution.

3.5 A golf putting problem

This problem was presented by S.M. Alessandrini [1] as a motivational example for teaching numerical methods for two-point boundary value problems (BVPs). With Daets, we have to treat it as an IVP rather than a BVP, but the program below can be used to explore solving the BVP by means of a "manual shooting method'.

A golf ball lies on a green, at (x_0, y_0) (in plan view, ignoring the z co-ordinate for now) and we wish to hit it to fall into the hole at $(0, 0)$. The green is not necessarily flat, and its surface is described by a function

$$0 = g(x, y, z) = z - S(x, y). \tag{3.6}$$

The rolling dynamics of the ball is ignored: it is regarded as a point sliding on a surface with coefficient of friction μ. It is confined to the surface, so no matter how curved the green, the ball can never fly into the air. This is a valid assumption if, and only if, the motion is slow enough and the curvature small enough that the normal reaction force $\mathbf{R}$ of surface on ball has a positive z component.

In [1] the system is modeled as an ODE, via the assumption that the magnitude R of $\mathbf{R}$ equals the component, normal to the surface, of the ball's weight. This gives a reasonable approximation, but when one solves it using the MATLAB code BVP4C and does a rotatable 3D plot, its deficiencies are obvious. Where the green is appreciably curved, the ball spends most of its time in the air or underground! A correct formulation is as a DAE, which is a typical index 3 mechanical system, like (1.2) except for the presence of friction. The reaction R acts as a Lagrange multiplier, like λ in (1.2).

Let $\mathbf{x} = (x, y, z)$ be the position of the ball. Taking the mass of the ball as unity, one may write the equations of motion as

$$\ddot{\mathbf{x}} = -\mu R \mathbf{n}_v + R\mathbf{n}_S - G\mathbf{k}, \tag{3.7}$$

where "dot" is $\mathrm{d}/\mathrm{d}t$; $\mathbf{n}_v$ is the unit vector tangential to the surface in the direction of the velocity $\dot{\mathbf{x}}$; $\mathbf{n}_S$ is the unit normal to the surface in the upward direction; G is the acceleration of gravity; and $\mathbf{k}$ is the unit vector vertically upward. Thus

$$\mathbf{n}_v = \frac{\dot{\mathbf{x}}}{\|\dot{\mathbf{x}}\|}, \quad \text{and} \quad \mathbf{n}_S = \frac{\nabla g(\mathbf{x})}{\|\nabla g(\mathbf{x})\|}. \tag{3.8}$$

The DAE consists of (3.6, 3.7) after substituting the formulae (3.8) into (3.7).

It is physically clear there are 4 DOF: the initial (x, y) and $(\dot{x}, \dot{y})$. Formulating as a BVP, we remove 2 DOF by specifying $(x, y) = (x_0, y_0)$ at $t = 0$. Specifying that the path reach the hole at some *chosen* time t_{end} would remove the remaining 2 DOF, but this is not realistic: a golfer does not choose t_{end}, but aims to make the ball's speed nearly zero at the moment it reaches the hole, so that it does not jump the hole. So, we treat t_{end} as an extra unknown, and remove the extra DOF by the condition $\|\dot{\mathbf{x}}(t_{\text{end}})\| = 0$. A MATLAB formulation of this (as an ODE system) has been successfully solved by BVP4C for a number of shapes of green. A difficulty—no doubt obvious to a golfer—is that if the slope of the green near the hole is so large that gravity overcomes friction, there may be no solution.

We give a crude program that solves the DAE as an IVP. A difficulty is that $\mathbf{n}_v$ in (3.7) creates a singularity when the ball comes to rest. If one takes no precautions over this the integration oscillates indefinitely—to see why, try solving the one-variable ODE $y' = -\text{sign}(y)$, $y(0) = -1$, which shows essentially the same behavior, with any standard ODE solver.

To overcome this, the program solves in one-step mode and terminates the integration when it finds $(\dot{x}, \dot{y})$ this step is so different from $(\dot{x}, \dot{y})$ at the last step, that their dot product is < 0. Another useful technique is shown: output (of $t, x, y, z, \dot{x}, \dot{y}, \dot{z}$) is sent to a file, not at each step but at equal time intervals of δt, so that one can estimate from a graph how the speed is changing along the path.

```
/* file golfputt.cc in daets/examples
   This program integrates the "golf putt" problem from Alessandrini 1995.
   Function outputfcn writes to a file.
   Line 1 contains the parameters that define the green. The rest of the
   file contains t,x,y,z,x',y',z' values suitable for Gnuplot.
*/
#include "DAEsolver.h"
using namespace daets;
/*-----------------DEFINE SURFACE OF GREEN & ITS GRADIENT-------------*/
template <typename T>
void dfgreen(const T &x, const T &y, const T &z,
             void *param,
             T &f, T &dfx, T &dfy, T &dfz) {
  /*Compute both f and grad f where f(x,y,z)=0 defines the surface
    of the green. f must be defined so that df/dz is positive.
    Input:  (x,y,z).
    Output: f              holds f(x,y,z),
            (dfx,dfy,dfz) holds grad f.
  */
  // This green has the form z - (f1 + f2) = 0 where f1 defines a plane
  // and f2 defines a Gaussian hump centered at (x,y)=(a,b) of height h
  // and "width squared" v.
  double *p = (double*)param;
  double dgx = p[0], dgy = p[1],
         a = p[2], b = p[3], h = p[4], v = p[5];
  T f1 = (dgx*x + dgy*y);                              // planar part
  T f2 = h * exp( -(sqr(x-a) + sqr(y-b))/(2*v) ); //hump part
  f = z - (f1 + f2);
  dfx = - ( dgx - (x-a)/v*f2 );
  dfy = - ( dgy - (y-b)/v*f2 );
  dfz = 1;
}
/*------------------------FCN, DEFINES EQNS OF MOTION-----------------*/
template <typename T>
void fcn(T t, const T *w, T *f, void *param) {
  // state variables
  #define R w[0] // normal reaction force of green on ball.
  #define x w[1] // ]
  #define y w[2] // |, co-ordinates of ball on surface of green.
  #define z w[3] // ] z points upward.
  const double // constants from problem definition
    G  = 9.81, // gravity, in SI units
    mu = 0.04; // coefficient of friction between green and ball
  // velocity
  T vx = Diff(x,1);
  T vy = Diff(y,1);
  T vz = Diff(z,1);
  T v = norm2(vx,vy,vz);
  // vector normal to green
  T fval, dfx, dfy, dfz;
  dfgreen(x,y,z, param, fval,dfx,dfy,dfz);
  T normdf = norm2(dfx,dfy,dfz);
  f[0] = fval;
  f[1] = -Diff(x,2) + R*dfx/normdf - mu*R*vx/v;
  f[2] = -Diff(y,2) + R*dfy/normdf - mu*R*vy/v;
  f[3] = -Diff(z,2) + R*dfz/normdf - mu*R*vz/v - G;
  #undef R
  #undef x
  #undef y
  #undef z
}
/*--------------------------AUXILIARY FUNCTIONS-----------------------*/
template <typename T>
T norm2 (const T x, const T y, const T z) {
  return sqrt(sqr(x)+sqr(y)+sqr(z));
}
void outputfcn(FILE *fd, const DAEsolution & x) {
  fprintf(fd, "%.4e   % .6e  % .6e  % .6e  % .6e  % .6e  % .6e\n",
          x.getT(), x.getX(1,0), x.getX(2,0), x.getX(3,0),
          x.getX(1,1), x.getX(2,1), x.getX(3,1));
}
/*------------------------------MAIN PROGRAM---------------------------*/
int main(int argc, char* argv[]) {
  // Check correct no. of inputs on command line
  if (argc!=4) {
    cout << "\n*** Usage: " << string(argv[0]) << " u0 v0 dt\n"
         <<   "     where  u0, v0 are initial x and y velocities,\n"
         <<   "            dt is time interval for output.\n\n";
    exit(1);
  }
  const int n = 4; // size of the problem
  // Use atof() to convert real-number literals to double
  const double
    t0 = 0.0,             // initial time
    vx0  = atof(argv[1]), // initial x velocity
    vy0  = atof(argv[2]), // initial y velocity
    dt = atof(argv[3]);   // at intervals of dt, output to file
  double greenparams[] =
     {0, 0.035, // plane, slightly uphill in y direction
      3, 1.1,  // center of hump
      0.1, 0.25};// height and (width)^2 of hump
  DAEsolver Solver(n, DAE_FCN(fcn), greenparams);//create solver, analyze DAE
  Solver.printDAEinfo();
  Solver.printDAEtableau();
  Solver.printDAEpointStructure();
  DAEsolution x(Solver); // create solution object
  // initial position (x,y)=(0,0) with a guessed value z=0.
  // initial velocity (x',y')=(vx0,vy0) with a guessed value z'=0.
  x.setT(t0)
    .setX(1,0, 0.0,Fixed).setX(2,0, 0.0,Fixed).setX(3,0, 0.0)
    .setX(1,1, vx0,Fixed).setX(2,1, vy0,Fixed).setX(3,1, 0.0);
  // open output file:
  FILE *fd = fopen("golfputt.out", "w");
  // Its first line holds parameters defining the surface:
  for (int i=0; i<6; i++) fprintf(fd, "  %.6e", greenparams[i]);
  fprintf(fd, "\n");
  // declare variables to record change of direction of motion:
  double vx, vy;
  double vxold=vx0, vyold=vy0, v_vold=1;
  // do integration in one-step mode, upper limit 60 seconds:
  SolverExitFlag flag = success;
  x.setOneStepMode(true);
  double tend = t0 + dt;
  Solver.integrate(x, t0, flag);  // to output initial consistent point
  outputfcn(fd,x);
  while (flag==success && v_vold>0  && tend<60) { // 60 sec time limit!
    Solver.integrate(x, tend, flag);
    if (x.getT()==tend) {
      outputfcn(fd,x);
      tend = tend + dt;
    }
    vx = x.getX(1,1); vy = x.getX(2,1);
    v_vold = vx*vxold + vy*vyold;
    vxold = vx; vyold = vy;
  }
  outputfcn(fd,x);
  fclose(fd);
  // output results to screen
  x.printStats();
  x.printSolution();
  if (flag!=success) {printSolverExitFlag(flag); return 1;}
  return 0;
}
```

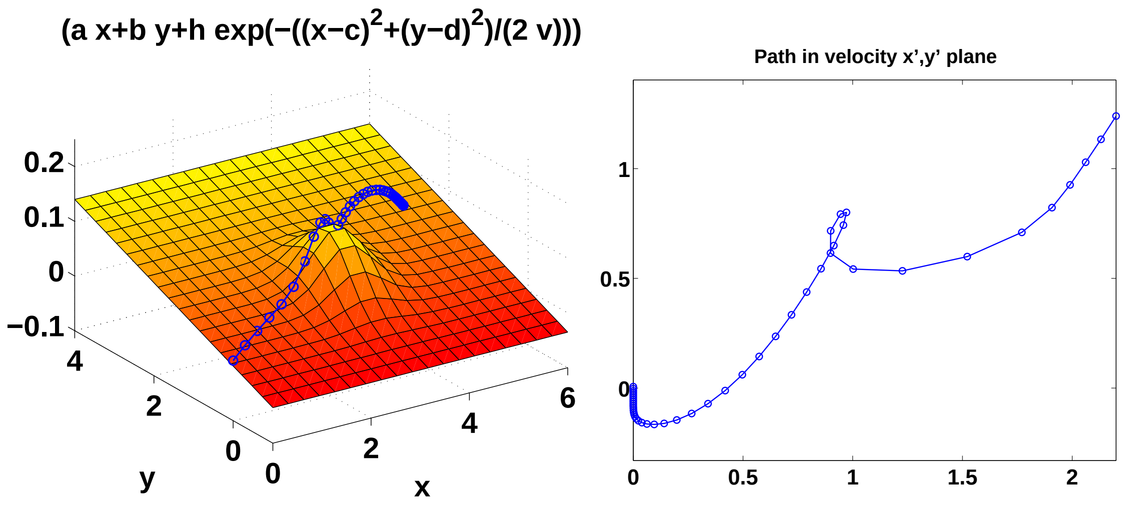


Figure 3.3: Golf putt plots. Markers are at 0.2 sec intervals. Vertical axis not to scale. (a) Path in x, y, z space. (b) Path in velocity $\dot{x}, \dot{y}$ plane.

The program displays the expected structural information:

```
DAE
---

      quasilinear
      size..................4
      index.................3

TABLEAU
-------

         |  0    1    2    3   |c_i
       -------------------------
        0|  -    0    0*   0   | 2
        1|  0    2*   1    1   | 0
        2|  0*   1    2    1   | 0
        3|  0    1    1    2*  | 0
       -------------------------
      d_j|  0    2    2    2   |

         Index = 3,  DOF = 4

POINT STRUCTURE
---------------
      variable   derivatives
         x0         -
         x1         0    1
         x2         0    1
         x3         0    1
```

The program as given uses SI units. It has a planar green with an uphill slope of 1:20 in the y direction, and a “Gaussian” circular hump of height 10cm and nominal radius 50cm, centered at the point $(3, 1.1)$m. The start point $(x_0, y_0) = (0, 0)$ is hardwired in the code, and the initial $(\dot{x}_0, \dot{y}_0)$ and the interval δt are read from the command line. The output for the graphs came from the call

```
./golfputt 2.2 1.24 .2
```

Figure 3.3 shows plots of the results, which show the ball being deflected uphill as it passes near the top of the hump, before it curls round and stops. They strongly suggest (what theory confirms) that the ball's velocity is directly downhill at the moment it comes to rest.

The MATLAB script used to produce the plots is given below.

```
curve=textscan(fid,'%n %n %n %n %n %n %n'); % read rest
fclose(fid);
[a,b,c,d,h,v] = deal(params{:}); % green's parameters
[t,x,y,z,xp,yp,zp] = deal(curve{:}); % curve data
figure(3)
plot(xp,yp,'o-','LineWidth',1.1)
set(gca,'FontSize',20,'FontWeight','bold', 'YTick',0:0.5:2);
axis equal
title('Path in velocity x'',y'' plane', ...
  'FontSize',18,'FontWeight','bold')
print -depsc golfputt3.eps
figure(4)
clf; hold on
set(gca,'FontSize',20,'FontWeight','bold' ...
       ,'CameraPosition', [-24 -32 2.2] ...
       , 'DataAspectRatio', [15 15 2]);
colormap(autumn);
wid = 3*sqrt(v);
fgreen=@(x,y)(a*x + b*y + h*exp(-((x-c).^2+(y-d).^2)/(2*v)) );
ezsurf(fgreen, ...
  [min([x;c-wid]) max([x;c+wid]) min([y;d-wid]) max([y;d+wid])], ...
  25);
plot3(x,y,fgreen(x,y)+0.001,'bo-','LineWidth',1.1)
print -depsc golfputt4.eps
```

3.6 Setting an output function

We illustrate how an output function is given to `integrate`, so it can output solution data into a file without calling the one-step mode. The example from Figure 1.2 is augmented in the listing of Figure 3.4, where line 23 sets the output function `outfcn`, defined in lines 32–39. On each successful step, `integrate` passes to this function the accepted solution and the **void*** parameters. Here, we output t, x, x''', y, y''', λ, λ'''. Plots of these variables and derivatives versus t are given in Figure 3.6.

3.7 Summary

This chapter has aimed to show typical techniques in the application of DAETS. The authors welcome suggestions for improving these examples, and contributions that show other useful techniques.

```
// file pendulumsimple_outfcn.cc in daets/examples
#include "DAEsolver.h"
using namespace daets;
template <typename T>
void fcn(T t, const T *z, T *f, void *param) {
  const double G = 9.8, L = 10.0;
  f[0] = Diff(z[0],2) + z[0]*z[2];
  f[1] = Diff(z[1],2)  + z[1]*z[2] - G;
  f[2] = sqr(z[0]) + sqr(z[1]) - sqr(L);
}
// Forward declaration of output function
void outfcn(const DAEsolution &x, void *output_param,
            void *problem_param);
int main() {
  const int n = 3;
  double t0 = 0.0, tend = 20.0;
  DAEsolver Solver(n, DAE_FCN(fcn));
  DAEsolution x(Solver);
  x.setT(t0)
    .setX(0,0, -9).setX(1,0, 0.0)
    .setX(0,1,0.0).setX(1,1, 1.0);
  FILE *outfile = fopen("pend.out", "w");
  x.setOutputFunction(outfcn, outfile);      // set output function
  SolverExitFlag flag;
  Solver.integrate(x, tend, flag);
  if (flag!=success)
    printSolverExitFlag(flag);
  fclose(outfile);
  return 0;
}
// Output function
void outfcn(const DAEsolution &x, void *out_param, void *problem_param) {
  FILE   *outfile = (FILE*)out_param;      // convert param to FILE* */
  fprintf(outfile, "%e   ", x.getT());     // output t
  // output each variable and its third derivative
  for (int j = 0; j < x.getNumVariables(); j++ )
    fprintf(outfile, "%e %e    ", x.getX(j,0), x.getX(j,3) );
  fprintf(outfile, " \n");
}
```

Figure 3.4: Setting an output function.

.

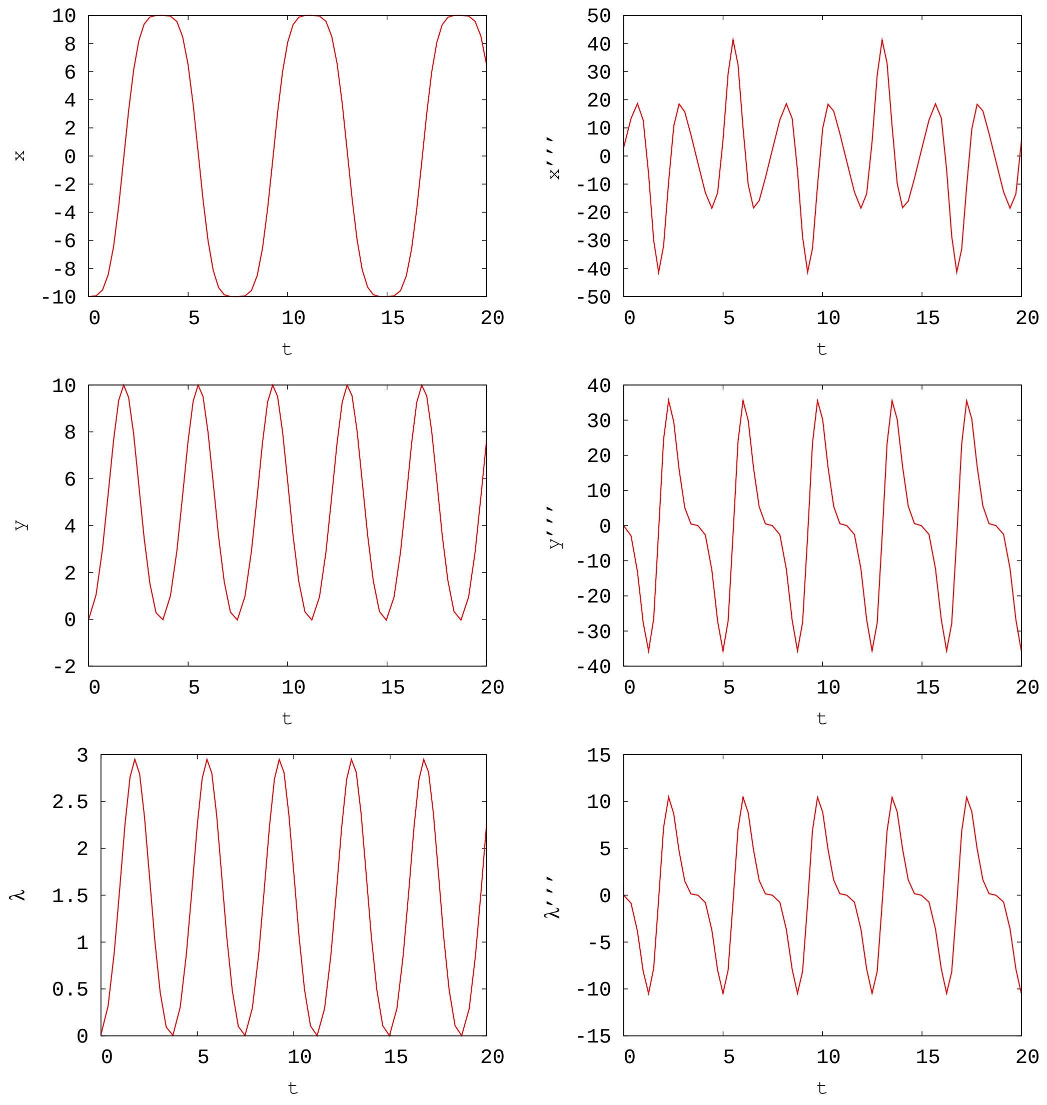


Figure 3.5: Plots of $x, x''', y, y''', \lambda$, and λ''' versus t for the simple pendulum.

Chapter 4

Installation

Read this chapter if DAETS is not already installed on your system. Section 4.1 describes how DAETS can be obtained, §4.2 describes the installation process, and §4.2.3 explains how the examples can be executed. The lists below show the supported platforms at the time of writing. Visit the web sites mentioned, for information on the current status of support.

In our experience, the main obstacles to installation come from the third-party components, mainly IPOPT. This is not a criticism of IPOPT, which is a powerful and inevitably somewhat complex optimization package. See if it is already on your system, in which case it may be possible to link to it instead of re-installing.

4.1 Obtaining DAETS

There are demo and full versions of DAETS. They are distributed as a set of `*.h` files and a precompiled library `libdaets.a`. These files are stored in zipped files as described below.

4.1.1 Demo version.

A free, demo version of DAETS is available at

`http://www.cas.mcmaster.ca/~nedialk/daets`

This version is restricted to solving systems of at most 8 equations. It is distributed as zipped files:

Architecture	OS	Compiler	Zip file
	Cygwin		`daets-1.1-demo-x86-cygwin-gcc.zip`
x86	Linux	GNU	`daets-1.1-demo-x86-linux-gcc.zip`
	Mac OSX		`daets-1.1-demo-x86-osx-gcc.zip`

4.1.2 Full version.

The full version is available through Canada's innovation portal, Flintbox, at `http://www.flintbox.com`

It is distributed in the following zipped files:

Architecture	OS	Compiler	Zip file
x86	Cygwin	GNU	`daets-1.1-x86-cygwin-gcc.zip`
	Linux		`daets-1.1-x86-linux-gcc.zip`
	Mac OSX		`daets-1.1-x86-osx-gcc.zip`
x86-64	Linux	GNU	`daets-1.1-x86-64-linux-gcc.zip`
		PathScale	`daets-1.1-x86-64-linux-pathCC.zip`
PowerPC	Mac OSX	GNU	`daets-1.1-ppc-osx-gcc.zip`

For other combinations of OS, architecture, compiler, and suggestions on improving DAETS, please contact

Ned Nedialkov	ned.nedialkov@reliablenumerics.com
John Pryce	j.d.pryce@ntlworld.com

4.2 Installation

After downloading the corresponding zip file, unzip it by e.g. `unzip`. This will create a directory `daets` with subdirectories:

subdirectory	contains
`include`	`*.h` files
`lib`	`libdaets.a`
`examples`	source code of the examples in this guide and the `makefile` used to produce the executables

The DAETS code uses the third-party components BLAS [10], LAPACK [11], FADBAD++ [21], IPOPT [22], and LAP [9]. The LAP program is distributed with the source code of DAETS. We describe how the rest are installed, and show how programs using DAETS are compiled and linked.

We use the GNU `make` utility.

4.2.1 Third-party components

LAPACK and BLAS

The LAPACK and BLAS libraries are likely installed on your machine. If not, you can download and install them from `http://www.netlib.org/lapack` and `http://www.netlib.org/blas`, respectively.

FADBAD++

1. Download FADBAD++ from `http://www.fadbad.com/fadbad.html`.
2. Extract the FADBAD++ files by

   ```
   unzip FADBAD++-2.1.zip
   ```

   These files will be stored in subdirectory `FADBAD++` of the directory in which you have executed `tar`.

IPOPT

1. Download the Fortran version `Ipopt-Fortran_2006Oct21.tgz` of the IPOPT package from `http://www.coin-or.org/download/source/Ipopt-Fortran/`
2. Extract the IPOPT files by

   ```
   tar -zxvf Ipopt-Fortran_2006Oct21.tgz
   ```

   This will create a directory `Ipopt-Fortran_2006Oct21` containing the IPOPT files.
3. Download `mc19ad.f`, `ma27ad.f`, and `ma47ad.f` from `http://www.cse.clrc.ac.uk/nag/hsl/` and save them in `Ipopt-Fortran_2006Oct21/OTHERS/HSL`.

   For an explanation of how to download these files, see `Ipopt-Fortran_2006Oct21/OTHERS/HSL/INSTALL.HSL`
4. Download `d1mach.f` from `http://www.netlib.org/blas/d1mach.f` and save it in `Ipopt-Fortran_2006Oct21/OTHERS/blas`
5. In `Ipopt-Fortran_2006Oct21`, type

   ```
   ./configure --prefix=DIRECTORY
   ```

   where `DIRECTORY` is the directory in which IPOPT is to be installed. For example, if you wish to install it in subdirectory `ipopt` of your home directory, you can type

   ```
   ./configure --prefix=$HOME/ipopt
   ```
6. In `Ipopt-Fortran_2006Oct21`, type

   ```
   make install
   ```

   This should install IPOPT in `DIRECTORY`.

   For more information about the `configure` options type

   ```
   ./configure --help
   ```

4.2.2 Creating executables

Here is the makefile (in `daets/examples`) used to produce the executables whose results are reported in this user guide.

```
# file makefile in daets/examples
# This file is used to produce the results in the
# DAETS user guide. We use g++ on an x86 MacOSX machine.
# SET C++ COMPILER AND FLAGS.
CXX    = g++
CXXFLAGS = -g -Wall -ansi
# SET DIRECTORIES
#   directory where DAETS is unpacked.
DAETS   = $(HOME)/daets
#   directory where FADBAD++ is unpacked.
FADBAD = $(HOME)/FADBAD++
#   IPOPT include directory
IPOPT_INCLUDE  = $(HOME)/ipopt/include
#   IPOPT lib directory, where libipopt.a is located
IPOPT_LIB  = $(HOME)/ipopt/lib
CXXFLAGS += -I$(DAETS)/include -I$(FADBAD) -I$(IPOPT_INCLUDE)
LDFLAGS  += -L$(DAETS)/lib -L$(IPOPT_LIB)
# LIBRARIES
LDLIBS   = -ldaets  -lipopt -llapack -lblas -lg2c -lm
# IF THE LINKING GIVES UNDEFINED REFERENCES HAVING _gfortran DO
# LDLIBS += -lgfortran
# FOR MAC OSX, YOU MAY HAVE TO ADD
# LDFLAGS += -bind_at_load
all: examples
EXAMPLES = chemakzo chemakzoUnknown golfputt pendulumshow_sa    \
        pendulumsimple pendulumsimple_outfcn laynewatson vdp
examples    : $(EXAMPLES)
clean:
        @- rm -rf *.out *.o core* *.bak $(EXAMPLES)
```

This file is self explanatory. We make the following comments. If the linker cannot find the LAPACK and BLAS libraries, you need to add to `LDFLAGS` the paths to them.

The `g2c` GNU FORTRAN runtime library (in `LDLIBS`) is normally present on the systems on which we have compiled DAETS. Install it if it is not on your system, and if you use GNU compilers. If you compile, e.g. the PathScale C++ [18] compiler, `libg2c.a` is not needed.

4.2.3 Running the examples

We recommend that you execute the programs in the `examples` directory. After you edit the `makefile` in this directory, type `make`. This will create the executable files `pendulumsimple` (§1.2), `pendulumshow_sa` (§2.1.2), `chemakzoUnknown` (§3.1), `chemakzo` (§3.2), `vdp` (§3.3), `laynewatson` (§3.4), and `golfputt` (§3.5).

In subdirectory `gnuplot` of `examples`, there are the files `vdp.gp` and `laynewatson.gp`, used to generate the plots in Figures 3.3 and 3.2, respectively. The executable `vdp` produces the data files `vdp-1.0e+00.out`, `vdp-1.0e+01.out` and `vdp-1.0e+02.out` in the current directory, from which you can generate the plots in Figure 3.3 by

```
gnuplot gnuplot/vdp.gp
```

Similarly, `laynewatson` produces the data file `laynewatson2.out`, from which the plot in Figure 3.2 can be generated by

```
gnuplot gnuplot/laynewatson.gp
```

Finally, `golfputt` produces `golfputt.out`, and you can generate plots using MATLAB and the program given in `examples/matlab/golfputtOutStudy.m`.

If you encounter discrepancies between the numerical results in this user guide and the output of your program executions, please contact the authors.

Chapter 5

Further theory. FAQs

This chapter is in two parts. Read Section 5.1 if you need more detail about the structural analysis theory on which DAETS is based, and references to the literature. Section 5.2 lists some—in our experience—frequently asked questions (FAQs) about the structural analysis approach, and about DAEs in general. The authors welcome suggestions for useful additions to the FAQs.

5.1 Structural analysis

Often, DAEs are generated in an exploratory way by simulation software that allows basic components (mechanical, electrical, etc.) to be connected using a graphical interface. It may be non-obvious what the resulting DAE requires as initial data, what its index is, and even whether it is ill-posed because it contains too little, or contradictory, information.

The structural analysis (SA) performed by DAETS helps to resolve such difficulties. Here we give enough theory that a user can, on small problems, do the analysis by hand. For further details see [14, 15, 19]. Our method gives essentially the same result as does, e.g., that of Pantelides [17], but is easier to use. Currently, several simulation packages offer SA facilities for model checking.

The steps to perform SA are as follows.

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

$$\sigma_{ij} = \begin{cases} \text{order of the derivative to which the } j\text{th variable } x_j \\ \text{occurs in the } i\text{th equation } f_i\text{; or} \\ -\infty \text{ if } x_j \text{ does not occur in } f_i. \end{cases} \tag{5.1}$$

2. A *transversal* T of Σ is a set of n positions in the matrix with one entry in each row and each column. That is, these positions can be put on the main diagonal by suitably permuting the columns (or rows). Its *value* is the sum of the σ_{ij} in those positions. Find a *highest value transversal* (HVT), one that makes the value as large as possible.

   For a well-posed DAE the value must be finite (hence an integer ≥ 0). Otherwise, it is $-\infty$, which happens if there does not exist a transversal all of whose σ_{ij} are finite. The DAE is then (structurally) *ill-posed* — there is some error in the problem formulation.

3. Find the *offsets*, integer vectors $\mathbf{c} = (c_1, \dots, c_n)$ and $\mathbf{d} = (d_1, \dots, d_n)$, with all $c_i \geq 0$, that satisfy

$$d_j - c_i \geq \sigma_{ij} \quad \text{for all } \ i, j \ \text{and} \tag{5.2}$$

with equality holding on the HVT. They are not unique, but there exist canonical *smallest* offsets, in the sense of $\mathbf{a} \leq \mathbf{b}$ if $a_i \leq b_i$ for each i. There is a simple method to find these:

Algorithm (Finding the offsets).
Input:

```
     Signature matrix $\Sigma$, and a transversal $T$ that must be a HVT
1.   The initial guess for $d_j$ is the largest $\sigma_{ij}$ in column $j$ of $\Sigma$
     Write $d_j$ beneath column $j$, for each $j$
2.   **do**
3.       Set each $c_i$ so that $d_j - c_i = \sigma_{ij}$ holds on $T$
         Write $c_i$ to the right of row $i$ of $\Sigma$, for each $i$
4.       Increase each $d_j$, where needed, by the least amount that makes
         $d_j - c_i \geq \sigma_{ij}$ hold everywhere
5.   **until** nothing changes
```

The c_i and d_j can only increase during the loop. The algorithm terminates if, and only if, T really is a HVT.

We like to show the results by a "signature tableau", which is Σ annotated with the offsets c_i, d_j and the names of the functions and variables and the positions of a HVT. We like to write $-\infty$ entries as a dash $-$ for readability: in larger systems typically almost all entries are $-\infty$. Also these are "forbidden" positions: a HVT of a well-posed DAE cannot have an entry in a $-\infty$ position.

This is illustrated below for the pendulum example, which has two HVTs, marked $^\bullet$ and $^\circ$.

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

It is easily seen that the above algorithm only needs one iteration to find the offsets of this signature matrix.

Also shown is the number F of degrees of freedom (DOF), given by

$$F = (\text{value of the HVT}) = \sum d_j - \sum c_i. \tag{5.3}$$

It is the number of *independent* IVs it requires, which is the same as the maximum number of IVs that can be specified "fixed", see `setX` in §1.3.3.

For n up to about 8, setting up Σ and finding a HVT "by eye" is usually easy. Otherwise, let DAETS do the analysis as suggested in §3.1.

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

$$\mathbf{J} = \frac{\partial\left(f_1^{(c_1)}, \ldots, f_n^{(c_n)}\right)}{\partial\left(x_1^{(d_1)}, \ldots, x_n^{(d_n)}\right)}, \quad \text{equivalently } \mathbf{J}_{ij} \; = \; \begin{cases} \dfrac{\partial f_i}{\partial x_j^{(\sigma_{ij})}} & \text{if } d_j - c_i = \sigma_{ij}, \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

(Here the x_j and derivatives thereof are treated as unrelated independent variables within f_i. For instance if f_1 is $x_1 x_2' x_1''$, then $\partial f_1/\partial x_1^{(2)} = \partial f_1 / \partial x_1'' = x_1 x_2'$.)

If $\mathbf{J}$, thus defined, is not identically singular, the SA-based method almost certainly succeeds. To be precise, if there is a consistent point as defined below where $\mathbf{J}$ is nonsingular, a solution to the DAE exists through that point; it is locally analytic, and the DAETS algorithm (in exact arithmetic) can find its Taylor series to arbitrary order.

For the pendulum, (5.4) gives the system Jacobian

$$\mathbf{J} = \begin{bmatrix} \partial f/\partial x'' & 0 & \partial f/\partial \lambda \\ 0 & \partial g/\partial y'' & \partial g/\partial \lambda \\ \partial h/\partial x & \partial h/\partial y & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 2x & 2y & 0 \end{bmatrix}. \tag{5.5}$$

This is not identically singular since $\det \mathbf{J} = -2(x^2 + y^2)$. Indeed at a consistent point, from the third equation of (1.2), $\det \mathbf{J} = -2L^2 \neq 0$.

5. The SA method *sometimes fails* (see FAQ 5.2.3). However, it always succeeds on various common classes of system. These include: index 0 DAEs (implicit ODEs); semi-explicit index 1 and index 2 DAEs; DAEs with Hessenberg structure of arbitrary index of which a particular case is index 3 constrained mechanical systems $M\mathbf{x}'' + G^T\boldsymbol{\lambda} = \boldsymbol{\phi}(t)$; $g(x) = 0$, where G is the Jacobian of g; purely algebraic systems $\mathbf{f}(t, \mathbf{x}) = \mathbf{0}$ (continuation problems), and the arc-length formulation of these. That is, it succeeds in exact arithmetic; ill-conditioning of the System Jacobian may cause it to fail in finite precision.

6. An upper bound for the standard *differentiation index* ν_d is given by the *Taylor index*

$$\nu_T = \max_i c_i + \begin{cases} 1 & \text{if some } d_j \text{ is zero,} \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

For the commonest kinds of DAE, $\nu_T = \nu_d$, but the difference can be arbitrarily large, see FAQ 5.2.5.

7. It was explained in §1.2.2 that required IVs, which are in fact the values carried along from step to step throughout the integration process, comprise the vector

$$\mathbf{X} = \Big(x_1, x_1', \ldots, x_1^{(d_1+\alpha)};\ x_2, x_2', \ldots, x_2^{(d_2+\alpha)}\ ;\ \ldots\ ;\ x_n, x_n', \ldots, x_n^{(d_n+\alpha)}\Big), \tag{5.7}$$

where $\alpha = -1$ if the DAE is quasilinear, $\alpha = 0$ otherwise; and, for consistency, $\mathbf{X}$ must satisfy the set of equations

$$\mathbf{0} = \mathbf{F} = \Big(f_1, f_1', \ldots, f_1^{(c_1+\alpha)};\ f_2, f_2', \ldots, f_2^{(c_2+\alpha)}\ ;\ \ldots\ ;\ f_n, f_n', \ldots, f_n^{(c_n+\alpha)}\Big). \tag{5.8}$$

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

f_i' means $\mathrm{d}f_i/\mathrm{d}t$ treating the variables and their derivatives as (unknown) functions of t, for instance if $f_1 = x_1'' - x_1x_3$ then $f_1' = x_1''' - x_1'x_3 - x_1x_3'$; similarly for higher derivatives.

The reason for the extra components in the non-quasilinear case is as follows. Consider a particular independent variable value t. If a set of values $\mathbf{X}$ in (5.7) is consistent with *some* solution of the DAE at t, then in the linear case that solution is *unique*. In the nonlinear case, there may be *several* solutions consistent with these values; however, including the next level of derivatives in $\mathbf{X}$ restores uniqueness.

This affects the user at the initial point $t =$ `t`. The initial conditions you provide are usually only *approximately* consistent values, which the code corrects by a root-finding process. By providing (hopefully good guesses of) these extra derivatives you make it more likely, in cases of non-uniqueness, that the code finds the consistent values — hence the DAE solution — that you intended.

An example is the use of DAETS to do arc-length continuation (example in §3.4). Here t is arc-length from an initial point $\mathbf{X}_0$ along a path in some $\mathbb{R}^N$. There is inherent non-uniqueness: from $\mathbf{X}_0$ you can traverse the path in either direction. In this case the extra derivatives DAETS forces you to provide turn out to be a guess of the unit tangent at $\mathbf{X}_0$ in the direction you wish to go. In most cases, you can estimate this *a priori*.

5.2 Frequently asked questions

5.2.1 *How well does the accuracy tolerance control the accuracy?*

Our experiments, see for instance [16, Section 5.1], suggest that for most problems, the error in the solution is roughly proportional to the tolerance, if the type of control (relative, absolute or mixed) specified by the `esttype` argument of `setTol` is unchanged. Thus the behavior is essentially the same as for an ODE code. However, this relation between tolerance and accuracy achieved cannot be guaranteed. The user is strongly recommended to call DAETS with more than one value for `tol` and to compare the results obtained to estimate their accuracy. For instance `1e-7` and `1e-8` if 7 digits of accuracy are wanted.

5.2.2 *If the method underlying* DAETS *sometimes fails, can the code be trusted?*

Yes. If a nonsingular system Jacobian $\mathbf{J}$ is found at a consistent point, in exact arithmetic this constitutes a proof that the DAE is locally solvable and the DAETS method will work, see [19, Theorem 4.2] and [14, Theorem 4.2]. Then in finite precision, things can only go wrong because $\mathbf{J}$ is ill-conditioned, in which case we suspect any DAE method, given the same equations and the same precision, would have trouble.

5.2.3 *What remedies are there when SA fails?*

The SA method relies on the fact that DAEs from applications are typically sparse—i.e., just a few variables and derivatives occur in each equation. If the sparsity is destroyed, the signature matrix may not reflect the "underlying mathematical structure" of the DAE, and SA gives the wrong result. For a crude example, if the pendulum system $\mathbf{f} = \mathbf{0}$ is converted to the equivalent $M\mathbf{f} = \mathbf{0}$ where M is a random nonsingular constant 3×3 matrix, all three rows of Σ become $[2, 2, 0]$ and SA gives the wrong result.

In interesting cases the failure of SA is more subtle. Campbell and Griepentrog [3] give the DAE of a robot arm that is index 5 but seems to be index 3. The Ring Modulator in [12], in case $C_s = 0$, is an index 2 electrical circuit DAE that seems to be index 1.

In our experience this has always been curable by purely algebraic manipulation of the equations to make them "more sparse". For instance, in the robot arm DAE, one can make SA find the correct index by naming one common subexpression in the equations as a new algebraic variable. In the ring modulator DAE, suitably adding to one equation a linear combination of other equations "moves the sparsity around" to give the desired effect. We do not at present know any systematic way to do such things, or any theory about when it can be done.

5.2.4 *The signature matrix is sometimes wrong. Can this spoil the computed solution?*

The point here is that the order of derivative of each variable in each equation is found by a pass through the code list, treating it symbolically but without any "clever" algebraic rearrangement. So for instance if equation 1 is $x_1 x_3 + (t x_2')' - t x_2'' = 0$, the code does not notice that the x_2'' terms cancel, and sets σ_{12} to 2 instead of the correct value 1.

Such an error can overestimate but not underestimate any σ_{ij}. It is proved in [14, Theorems 5.1 and 5.2] that this cannot deceive the method. Just one of two things happens. Either the overestimation is "small", and the method computes the correct Taylor series but with some TCs possibly being computed a little later than they might have been (causing slight inefficiency). Or it is "large", and in exact arithmetic $\mathbf{J}$ is structurally singular. In the latter case, in finite precision $\mathbf{J}$ will be structurally singular within roundoff, a situation that is easily detected numerically.

5.2.5 *What about Reißig's example?*

Reißig *et al.* [20] give a family of DAEs for which the structural index can be an arbitrary $\nu > 1$, though the differentiation index is 1. For $\nu = 3$ the system is

$$\begin{aligned} \dot{x}_2 + \dot{x}_3 + x_1 &= a(t),\\ \dot{x}_2 + \dot{x}_3 + x_2 &= b(t),\\ \dot{x}_4 + \dot{x}_5 + x_3 &= c(t),\\ \dot{x}_4 + \dot{x}_5 + x_4 &= d(t),\\ x_5 &= e(t). \end{aligned}$$

Its signature matrix and system Jacobian are

$$\Sigma = \begin{bmatrix} 0 & 1 & 1 & - & - \\ - & 1 & 1 & - & - \\ - & - & 0 & 1 & 1 \\ - & - & - & 1 & 1 \\ - & - & - & - & 0 \end{bmatrix} \qquad \text{and} \qquad \mathbf{J} = \begin{bmatrix} 1 & 1 & 1 & & \\ & 1 & 1 & & \\ & & 1 & 1 & 1 \\ & & & 1 & 1 \\ & & & & 1 \end{bmatrix}.$$

One easily sees that the SA method succeeds, with $c_i = (0, 0, 1, 1, 2)$, $d_j = (0, 1, 1, 2, 2)$, and gives $\nu_T = 3$. But by inspection, one can write the general solution explicitly in a form involving no derivatives of the driving terms: sufficient proof that the index is 1.

In fact, using two new variables $x_6 = x_2 + x_3$ and $x_7 = x_4 + x_5$ to eliminate the common subexpressions $\dot{x}_2 + \dot{x}_3$ and $\dot{x}_4 + \dot{x}_5$, one obtains a system on which SA succeeds. It has all $c_i = 0$, and all $d_j = 0$ except $d_6 = d_7 = 1$.

This raises a different issue from either FAQ 5.2.3 or FAQ 5.2.4, but the "cure" by defining new variables can be seen as an illustration of the former.

5.2.6 *How can a leading derivative not actually occur in the DAE?*

Consider the classic example of Linda Petzold

$$\begin{aligned} 0 &= x_1 - g(t),\\ x_1' &= x_2,\\ x_2' &= x_3, \end{aligned}$$

where $g(t)$ is an arbitrary given function. The offsets are $d_j = 2, 1, 0$ and $c_i = 2, 1, 0$. Thus x_1'' is a leading derivative, though it does not occur in the system. By adding extra equations $x_3' = x_4,\ \ldots$ one can make an arbitrarily large discrepancy between the leading derivative order and the order that occurs in the equations.

5.2.7 *Not many standard functions are allowed in* `fcn`*. Are more planned?*

These have sufficed for the examples we have done so far. If you need more, let us know.

5.2.8 *Have you plans for an event-location facility?*

Yes, and efficient step-by-step output or plotting without using one-step mode, and banded/sparse linear algebra, and other features that modern ODE solvers have. Please show us some real applications and say what your needs are, so we can set our priorities.

5.2.9 *Can* DAETS *handle diagnostics silently?*

That is, can it be put in a mode where it handles exceptional conditions, not by printing a diagnostic message, but by returning a diagnostic data structure so that the calling program can take appropriate action? This allows the code to be used as a numerical "engine" at the heart of some application.

This will be in the next version. The exception-handling mechanism of C++ provides most of the needed features.

5.2.10 *Have you tips for debugging code that uses* DAETS*?*

One of us (JDP) is a C++ beginner. He has found that most of his troubles solving problems with DAETS have been C++ ones (like forgetting the `namespace` line at the start of a program). For specific DAE difficulties, the suggestions given in §3.1 have worked quite well. If you find some typical difficulties or traps in using DAETS, we shall welcome tips on how to avoid them, for possible inclusion in these FAQs.

5.2.11 *Can I have more derivatives printed?*

We plan this in the next version. One tip is that if you have a quasilinear problem, so the code only prints out derivatives up to order d_j-1, and you would like it to do so up to order d_j, you can do so by fooling the code into thinking it is not quasilinear. For instance, in the simple pendulum DAE you could add $\lambda^2 - \lambda^2$ (or $10^{-300} \times \lambda^2$) to one of the equations.

5.2.12 *How is the simple pendulum DAE derived?*

Let the pendulum be made of a light rigid rod of length L with a bob, which is a point of mass m, at the end. Let it be suspended from the origin O and take axes with x horizontal and y vertically downward. Let (x, y) be the position of the bob. Let T be the force in the rod, with tension being positive and compression negative. Let θ be the angle between the rod and the y axis, in the direction of positive x. Let G be gravity. Then Newton's second law gives the equations of motion:
Resolving horizontally,

$$mx'' = -T \sin\theta.$$

Resolving vertically,

$$my'' = -T \cos\theta + mG.$$

Now $\sin\theta = x/L$ and $\cos\theta = y/L$.

Substituting these into the the equations of motion, and defining $\lambda = T/(mL)$, gives

$$\begin{aligned} x'' + \lambda x &= 0, \\ y'' + \lambda y - g &= 0. \end{aligned}$$

The final equation is the rigidity constraint $x^2 + y^2 = L^2$.

5.2.13 *I'm new to DAEs, why are they so different from ODEs?*

No short explanation will be useful to everyone, but here's a try. Take the typical case where the independent variable t is time, and the dependent (=state) variables $x_1(t), x_2(t), \ldots$ describe how the state of some physical system evolves in time.

With an ODE, the rate of change of each state variable is explicitly given in terms of the state variables, and possibly t. If there is just one (scalar) state variable x, so that the ODE can be written $\mathrm{d}x/\mathrm{d}t = f(t, x)$, one can draw a picture in the (t, x) plane. Through any point, say (t_0, x_0), draw a line segment whose gradient is $f(t, x)$. This line is along the tangent to the solution of the ODE that passes through (t_0, x_0).

Drawing enough of these segments produces a *direction field* for the ODE, which gives a good feel for the shape of the solutions. One can join the segments up by eye to sketch solutions quite accurately. This is illustrated in Figure 5.1 for the ODE

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\frac{t}{x}.$$

It is clear from the direction field that the solutions are circles, and, indeed, the general solution of this ODE is easily seen to be $x^2 + t^2 = c$, where c is a constant.

The same goes for an ODE with several state variables $x_j(t)$, except that the line segments are now drawn in a multi-dimensional space. Most numerical methods

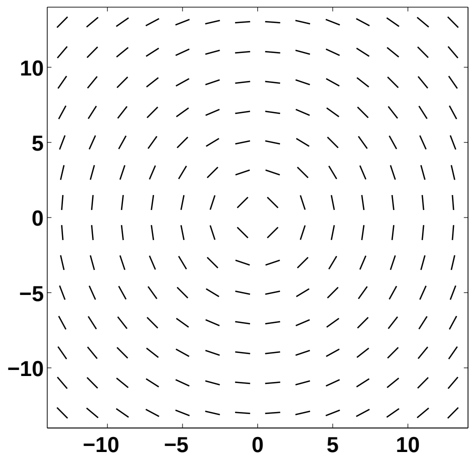


Figure 5.1: Direction field of $\mathrm{d}x/\mathrm{d}t = -t/x$.

for ODE initial value problems are essentially sophisticated ways of joining up little line segments.

DAEs are not like that, or rather, only partly like that. It may help to look at this from the physical, rather than the mathematical, viewpoint, by considering the derivation of the simple pendulum equations in FAQ 5.2.12.

There, it is clear that x and y are ODE-like variables—two equations define their acceleration, which governs their change in velocity, which governs their change in position. However, λ, which is a constant factor times the force T of tension/compression in the rod, is physically quite different. The value of T is not specified by a rate-of-change, but rather is an instantaneous response to the current position and velocity of the bob.

Thus it is not surprising that the mathematics shows x'', y'' and λ entangled, in that they are, at any instant, the solution of three simultaneous equations in which the known values of x, y, x', y' appear as constants.

It may help to think of the rod as being ever so slightly elastic. Thus, at any instant it stretches or compresses infinitesimally, to produce that precise force T on the bob which keeps it going in an exact circle. Were we to model the dynamics of this elasticity, the equations of motion would be an ODE. The DAE is the limiting, and simpler, case where the elasticity goes to zero.

For a purely mathematical illustration why DAEs are not ODEs, see the example DAE in FAQ 5.2.6. It has no degrees of freedom: its unique solution is $x_1 = g(t)$, $x_2 = g'(t)$, $x_3 = g''(t)$. But if the zero on the left of the first equation is replaced by $\epsilon x_3'$, for a nonzero ϵ, it is a normal (but unpleasant when ϵ is small) ODE with three degrees of freedom, requiring three IVs to specify a unique solution.

The pendulum also has a well known description by an ODE:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{G}{L}\sin(\theta),$$

where θ is as in FAQ 5.2.12. Many more complex mechanical systems, such as robotic arms, can be modeled by ODEs, typically involving angles of rotation.

Why use a DAE when an ODE is available? One good reason is that as the size of the system increases it becomes increasingly harder to formulate such ODE models, whereas the algorithm to formulate the DAE form typically remains straightforward. This is true in other application areas, such as electrical circuit modeling. For simulation packages that let one define a system interactively, and that then set up its governing equations automatically, this gives the DAE form a decided advantage.

Appendix A

Revision History

In this appendix, we report changes in the DAETS code since version 1.0 of June 2008.

A.1 Version 1.1, May 2009

This version has been extensively tested using the CPPUNIT [4] testing framework, and the code has been analyzed with the GCOV [5] test coverage tool. This work is reported in [24]. As a result, we have fixed several minor bugs in the code and inaccuracies in the user guide.

Changes in functionality compared to version 1.0 are as follows.

1. The `getX` method of the `DAEsolution` class can now return derivatives of an x_j up to $p + d_j$, that is $x_j^{(k)}$ for any $k \in \{0, \ldots, p + d_j\}$, where p is the order of the Taylor series; see p. 27.

2. We have added a mechanism for data output inside the `integrate` function. This allows, for example, one to generate a file with solution data suitable for plotting. See p. 30 and §3.6.

3. The constant in the formula for computing $h_{\min}$ (2.4) is changed from 10 to 16.

Index

Bibliography

[1] S. Alessandrini, *A motivational example for the numerical solution of two-point boundary-value problems*, SIAM Review, 37 (1995), pp. 423–427.

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

[3] S. L. Campbell and E. Griepentrog, *Solvability of general differential algebraic equations*, SIAM J. Sci. Comput., 16 (1995), pp. 257–270.

[4] *CppUnit project.* `http://sourceforge.net/projects/cppunit/`

[5] *Gcov—a test coverage program.* `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

[6] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers in applied mathematics, SIAM, Philadelphia, PA, 2000.

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

[8] Intel Corp., *Intel C++ compiler for Linux web page.* `http://support.intel.com/support/performancetools/c/linux`.

[9] R. Jonker and A. Volgenant, *A shortest augmenting path algorithm for dense and sparse linear assignment problems*, Computing, 38 (1987), pp. 325–340.

[10] LAPACK Project, *BLAS — Basic Linear Algebra Subprograms.* `http://www.netlib.org/blas/`.

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

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

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

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

[15] ——, *Solving differential-algebraic equations by Taylor series (II): Computing the System Jacobian*, BIT, 47 (2007), pp. 121–135.

[16] ——, *Solving differential-algebraic equations by Taylor series (III): The DAETS code*, J. Numerical Analysis, Industrial and Applied Mathematics, 3 (2008), pp. 61–68.

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

[18] *PathScale compiler suite.* `http://www.pathscale.com`.

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

[20] G. Reissig, W. S. Martinson, and P. I. Barton, *Differential–algebraic equations of index 1 may have an arbitrarily high structural index*, SIAM J. Sci. Comput., 21 (2000), pp. 1987–1990.

[21] O. Stauning and C. Bendtsen, *FADBAD++ web page.* FADBAD++ is available at `http://www.fadbad.com`.

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

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

[24] B. Zheng, *Testing scientific computing software. A case study of a DAE solver*, MSc thesis (in preparation), Dept. of Computing and Software, McMaster University, Hamilton, Ontario, Canada, L8S 4K1, 2009.