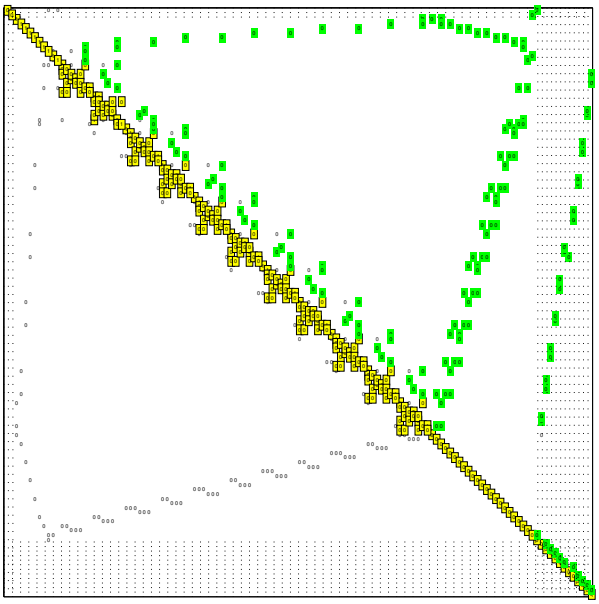
DAESA

User Guide

Version 1.0

Ross McKenzie
John D. Pryce
Cardiff University
United Kingdom

Guangning Tan
Nedialko S. Nedialkov
McMaster University
Canada



Preface

DAESA, Differential-Algebraic Equations Structural Analyzer, is a MATLAB tool for structural analysis (SA) of differential-algebraic equations (DAEs). It allows convenient translation of a DAE system into MATLAB and provides a small set of easy-to-use functions. DAESA can analyze systems that are fully nonlinear, high-index, and of any order. It determines structural index, degrees of freedom, constraints, variables to be initialized, and suggests a solution scheme.

The structure of a DAE can be readily visualized by this tool. It can also construct a block-triangular form of the DAE, which can be exploited to solve it efficiently in a block-wise manner.

This code was originated by JDP in 2001-2003. Ning Liu (M.Sc. 2006, Computing and Software, McMaster) added new functionality, and in particular the computation of signature matrices through operator overloading. GT did a major rewrite and added many new features to DAESA. Ian Washington (Ph.D. candidate, Chemical Engineering, McMaster) was the first user of DAESA—he applied it to study chemical engineering problems and suggested many improvements. The figure on the title page is from DAESA applied to one of these problems.

We are hoping that this tool will be helpful to researchers and practitioners when studying systems containing differential and algebraic equations. It could also be applied to pure differential and pure algebraic systems.

For bug reporting, questions, and suggestions on improving DAESA, please contact the authors at:

daesatool@gmail.com

We acknowledge with thanks the support given to RM and JDP by the Leverhulme Trust and the Engineering and Physical Sciences Research Council, both of the UK, and to GT and NSN by the Canadian Natural Sciences and Engineering Research Council and the McMaster Centre for Software Certification.

July 9, 2013

Contents

In the electronic version of this document, every cross-reference is a hyperlink. For instance you can click on the entry "Quick start" 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.

List of Figures

List of Tables

Chapter 1

DAESA Overview

DAESA, Differential-Algebraic Equations Structural Analyzer, is a MATLAB tool for structural analysis (SA) of differential-algebraic equations (DAEs). It allows convenient translation of a DAE into MATLAB and provides a set of (currently 18) easy-to-use functions for determining and visualizing key structural properties of the DAE.

The package is applicable to DAE systems of the general form

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

where t is the independent variable, and the $x_j(t)$ are n state variables. The formulation (1.1) includes high-order systems and systems that are nonlinear in leading derivatives. Furthermore, (1.1) includes systems of ordinary differential equations (ODEs) and pure algebraic systems.

In the next paragraphs, a concept is set in *slanting type* on first occurrence, with a forward reference to further details about it.

DAESA performs analysis that is similar to the one the C++ solver DAETS does [5, 6]. However, DAETS is not suitable for rapid investigation of DAEs, as it requires C++ knowledge and compiling the user code. DAESA is a light-weight, easy-to-use tool for SA that provides convenient facilities for analyzing a DAE to find its *structural index* (§3.1), the number of *degrees of freedom*, henceforth referred to as the *DOF* (§3.1), the *constraints* and required *initial values* (§3.5) and a *solution scheme* (§3.1), and also to reduce the DAE to a coarse or a fine *block-triangular form* (BTF) (§3.3), which can be exploited for efficient solution in a block-wise fashion.

DAESA applies MATLAB's operator overloading to process a DAE given by a user-supplied function for evaluating the f_i in (1.1). First, using operator overloading, DAESA extracts the DAE's *signature matrix* (§3.1), from whose sparsity pattern the coarse block-triangular form can be found. It then finds out if the problem is structurally well-posed, and if so, solves a linear assignment problem to calculate the (global) *offsets* (§3.1) of the DAE, which give the structural index and DOF (§3.1).

Using both the signature matrix and the offsets, DAESA constructs the fine block-triangular form, finds *local offsets* (§3.5) for the blocks, and for each block, determines if it is quasilinear in the leading derivatives (§3.5). Based on the local offsets and linearity information, DAESA deduces a minimal set of variables and derivatives of them that need to be initialized and a minimal set of inherent constraints.

The package provides functions for displaying the original sparsity structure of the DAE, as well as

for displaying its coarse and fine block-triangularizations, and functions for reporting the constraints, initialization summary, and a solution scheme for the DAE.

Remark 1 The structural index computed by DAESA is an upper bound on the differentiation index, and in our experience it usually equals it. That is, DAESA usually finds the smallest possible set of differentiations. Although successful on many problems of interest, the underlying SA theory (and DAESA respectively) may fail to determine the correct structural, and therefore differentiation index on some problems, see e.g. [4, 9].

We would have a "certificate" that the SA is successful, if the *system Jacobian* (§3.1) is non-singular at a consistent point, see also [4]. The present tool does not compute consistent points and does not evaluate the system Jacobian: it performs symbolic-type analysis of DAEs. We plan to incorporate the evaluation of this Jacobian in a next version of DAESA.

This user guide is organized as follows. Chapter 2 provides a quick start to using DAESA—we show basic SA with it. Chapter 3 gives an overview of the theory behind DAESA. Chapter 4 describes the functions of DAESA. Chapter 5 presents several examples of analyzing DAEs. How to obtain and install DAESA is described in Chapter 6.

Chapter 2

Quick start

In this chapter, we illustrate how DAESA performs basic SA (more examples are in §5). Using DAESA involves three steps:

1. specifying a DAE (§2.1),
2. calling the main SA function (§2.2), and
3. extracting structural data (§2.3).

2.1 Specify a DAE

A DAE is given by a function with arguments and return value as follows

```
function f = daefcn(t,x,options)
```

Here `t` and `x` are the independent and dependent variables, respectively, and `options` is an optional list of one or more arguments; `x` is a column n-vector, where `x(j)` is variable x_j, and `f` is a column n-vector with `f(i)` containing the evaluation of f_i.

We apply our tool on the (artificially) modified two-pendulum problem (MOD2PEND), an index-7, non-quasilinear DAE (see also [8]):

$$\begin{aligned}
0 &= f_1 = x'' + x\lambda \\
0 &= f_2 = y'' + y\lambda - G \\
0 &= f_3 = x^2 + y^2 - L^2 \\
0 &= f_4 = u'' + u\mu \\
0 &= f_5 = (v''')^2 + v\mu - G \\
0 &= f_6 = u^2 + v^2 - (L + c\lambda)^2 + \lambda''
\end{aligned} \tag{2.1}$$

In the original two-pendulum problem,

$$f_5 = v'' + v\mu - G \quad \text{and} \quad f_6 = u^2 + v^2 - (L + c\lambda)^2.$$

The state variables are x, y, λ, u, v, and μ; G is gravity, $L > 0$ is the length of the first pendulum, and $c > 0$ is a given constant.

Figure 2.1 shows an encoding of (2.1).

```
1  function f = modified2pendula(t,z,G,L,c)
2   x = z(1); y = z(2); la = z(3);
3   u = z(4); v = z(5); mu = z(6);
4   % first pendulum
5   f(1) = Dif(x,2)+x*la;          % x'' + xλ = 0
6   f(2) = Dif(y,2)+y*la-G;        % y'' + yλ - G = 0
7   f(3) = x^2+y^2-L^2;            % x^2 + y^2 - L^2 = 0
8   % modified second pendulum
9   f(4) = Dif(u,2)  +u*mu;        % u'' + uμ = 0
10  f(5) = Dif(v,3)^2+v*mu-G;      % (v''')^2 + vμ - G = 0
11  f(6) = u^2+v^2-(L+c*la)^2+Dif(la,2);
12                          % u^2 + v^2 - (L + cλ)^2 + λ'' = 0
13 end
```

Figure 2.1: DAESA function for evaluating (2.1).

The `Dif(var,k)` operator returns the `k`th derivative of variable `var`. Constants G, L, and c are passed as additional parameters. In lines 2–3, the input variables `z(i)` are renamed for better readability. Equations (2.1) are translated in lines 5–11.

Remark 2 Specifying a DAE in DAESA is similar to how ODEs are specified in MATLAB (e.g. for `ode45`) except that, instead of returning $\mathbf{y}'$ in $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$, we return a vector of the f_i's in (1.1).

2.2 Perform structural analysis

Structural analysis is performed by the `daeSA` function:

```
function sadata = daeSA(daefcn,n,options)
```

Here `daefcn` is a function for evaluating the DAE, `n` is the size of the problem, and `options` is an optional list of one or more arguments that are passed by `daeSA` to `daefcn`. The return object `sadata` encapsulates data obtained from analyzing the DAE represented in `daefcn`. It can be accessed by the functions described in §2.3 and §4.

For our problem, we set values for the size and the additional parameters and then call `daeSA`:

```
n = 6; G = 9.8; L = 1.0; c = 0.1;
sadata = daeSA(@modified2pendula,n,G,L,c);
```

We shall use this `sadata` in the examples that follow.

Remark 3 The result of the SA is not affected by the values of G, L, and c. However, we provide this mechanism for passing them to `daefcn`, as such constants matter when computing a consistent point, evaluating the system Jacobian, and integrating the DAE. DAESA does not perform these three tasks, but we plan to implement them in the future.

2.3 Extract structural analysis data

One can visualize the structure of a DAE and obtain SA data as follows.

2.3.1 Visualization

To display the DAE structure, we call

```
showStruct(sadata);
```

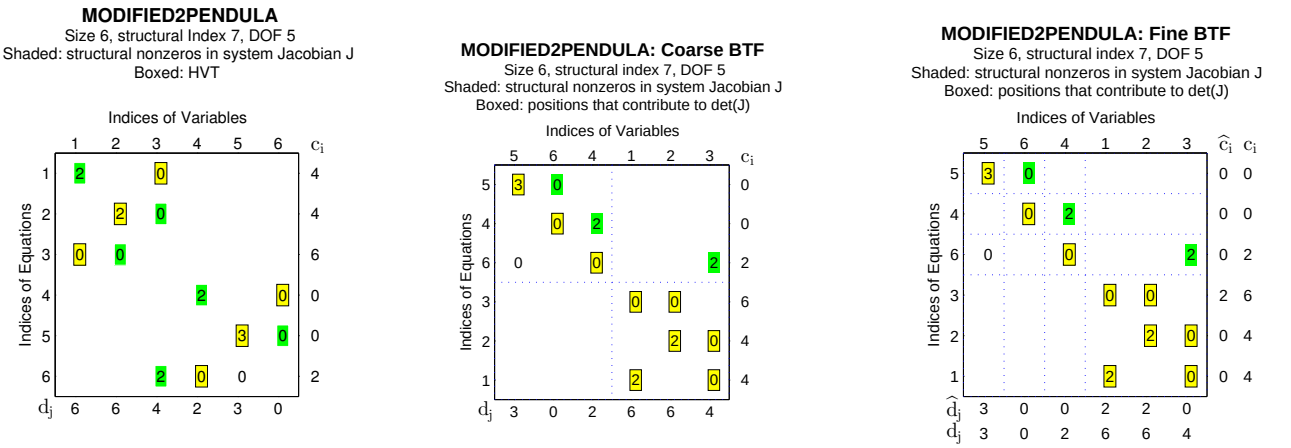
to produce the coarse block-triangularized DAE structure, we call

```
showStruct(sadata,'disptype','blocks');
```

and to produce the fine block-triangularized DAE structure, we call

```
showStruct(sadata,'disptype','fineblocks');
```

When called on `sadata` returned by `daeSA` above, these functions produce the plots in Figure 2.2.



(a) original structure (b) coarse block-triangularized structure (c) fine block-triangularized structure

Figure 2.2: Structure of (2.1) and its block-triangularizations.

In Figure 2.2 the $-\infty$ entries are not displayed. The top of the figures indicate index and DOF. The shaded entries (green and yellow in color print) are where the system Jacobian (§3.1), $\mathbf{J}$, is structurally nonzero. (a) The boxed entries denote a *highest-value transversal* (HVT) (§3.1); (b) and (c) the boxed entries denote the entries that contribute to $\det(\mathbf{J})$, which is the union of all HVTs [4]. Global offsets are denoted by c_i, d_j, and local offsets are denoted by $\widehat{c}_i$, $\widehat{d}_j$.

2.3.2 Index and DOF

The index and the DOF are obtained by

```
index = getIndex(sadata);
DOF = getDOF(sadata);
```

which return 7 and 5, respectively.

2.3.3 Initialization summary

The function `printInitData` reports which variables and derivatives of them need to be initialized for a numerical solution of the analyzed DAE. The call

```
vars = {'x','y','lam','u','v','mu'};
printInitData(sadata,'varnames',vars,'outfile','initdata.txt');
```

produces file `initdata.txt` with

```
modified2pendula problem
--------------------------------------------------------------------------------
Initialization summary:
x, x', y, y', u, v, v', v'', v'''
```

That is, x, x', y, y', u, v, v', v'', and v''' need to be initialized. By default, `printInitData` outputs as variable names `x` followed by a variable number from 1 to n. Here, these names are replaced using the key pair `'varnames'`, `vars`. If the key pair `'outfile'` and `'initdata.txt'` are omitted, the function will print the output to the command window.

2.3.4 Constraints

The DAE constraints are reported by the function `printConstr`. For example, the call

```
printConstr(sadata,'outfile','constr.txt');
```

produces file `constr.txt` with

```
modified2pendula problem
--------------------------------------------------------------------------------
Constraints:
f1, f1', f1'', f1''', f2, f2', f2'', f2''', f3, f3', f3'', f3''', f3'''',
f3^(5), f5, f6, f6', f6''
```

That is, the constraints are

$$\begin{aligned} f_1^{(c_1)} &= 0, \qquad c_1 \le 3, \\ f_2^{(c_2)} &= 0, \qquad c_2 \le 3, \\ f_3^{(c_3)} &= 0, \qquad c_3 \le 5, \\ f_5 &= 0, \qquad \text{and} \\ f_6^{(c_6)} &= 0, \qquad c_6 \le 2. \end{aligned}$$

The default equation-names `f1, f2, ...` can be changed, like the variable-names in §2.3.3, using the key `'fcnnames'` instead of `'varnames'`. Also, as in §2.3.3, omitting the key pair `'outfile'` and `'constr.txt'` will produce an output in the command window.

2.3.5 Solution scheme

A solution scheme is reported by `printSolScheme`. It reports how to solve the DAE (what variables to initialize, what equations to solve, and for which variables) by stages using its fine block-triangularized structure. In particular, it shows how to compute values for the derivatives of each variable.

Full solution scheme. The call

```
printSolScheme(sadata,'varnames',vars,...
                        'outfile','solscheme.txt','detail','full');
```

(with `vars` set as above) produces file `solscheme.txt` with

```
Solution scheme for 'modified2pendula' problem
--------------------------------------------------------------------------
Initialization summary:
x, x', y, y', u, v, v', v'', v'''
--------------------------------------------------------------------------
STAGE  k = -6, 1 block
- Block 4:6 -
    Solve nonlinear equation (give trial values)
    0 = f3  for  x, y
STAGE  k = -5, 1 block
- Block 4:6 -
    Using  x, y
    Solve linear equation (give trial values)
    0 = f3'  for  x', y'
STAGE  k = -4, 1 block
- Block 4:6 -
    Using  x, x', y, y'
    Solve linear 3x3 system
    0 = f1, f2, f3''  for  x'', y'', lam
```

```
STAGE  k = -3, 2 blocks
- Block 4:6 -
    Using  x, x', x'', y, y', y'', lam
    Solve linear 3x3 system
    0 = f1', f2', f3'''  for  x''', y''', lam'
- Block 1:1 -
    Solve nothing (give initial value)
    for    v
STAGE  k = -2, 3 blocks
- Block 4:6 -
    Using  x, x', x'', x''', y, y', y'', y''', lam, lam'
    Solve linear 3x3 system
    0 = f1'', f2'', f3''''  for  x'''', y'''', lam''
- Block 3:3 -
    Using  lam, lam', lam'', v
    Solve nonlinear equation (give trial value)
    0 = f6  for  u
- Block 1:1 -
    Solve nothing (give initial value)
    for    v'
STAGE  k = -1, 3 blocks
- Block 4:6 -
    Using  x, x', x'', x''', x'''', y, y', y'', y''', y'''', lam, lam',
           lam''
    Solve linear 3x3 system
    0 = f1''', f2''', f3^(5)  for  x^(5), y^(5), lam'''
- Block 3:3 -
    Using  lam, lam', lam'', lam''', u, v, v'
    Solve linear equation
    0 = f6'  for  u'
- Block 1:1 -
    Solve nothing (give initial value)
    for    v''
STAGE  k = 0, 4 blocks
- Block 4:6 -
    Using  x, x', x'', x''', x'''', x^(5), y, y', y'', y''', y'''', y^(5),
           lam, lam', lam'', lam'''
    Solve linear 3x3 system
    0 = f1'''', f2'''', f3^(6)  for  x^(6), y^(6), lam''''
- Block 3:3 -
    Using  lam, lam', lam'', lam''', lam'''', u, u', v, v', v''
    Solve linear equation
    0 = f6''  for  u''
- Block 2:2 -
    Using  u, u', u''
    Solve linear equation
```

```
    0 = f4  for  mu
- Block 1:1 -
    Using  v, v', v'', mu
    Solve nonlinear equation (give trial value)
    0 = f5  for  v'''
```

In this output, `Block p:q` denotes the sub-block in the *fine block-triangularized* DAE comprising rows `p` to `q` and columns `p` to `q`. For example, `Block 4:6` implies equations f_3, f_2, f_1 and variables x, y, λ; cf. Figure 2.2(c).

At the head of the output is an initialization summary, giving all the variables (and derivatives of them) that need initial values. When such a value is needed in the following solution process, it is marked by `(give trial value)`; when it is needed, but no equations are solved, it is marked by `(give initial value)`. Trial values are guesses by the user, which a solver may change to satisfy constraints; initial values are not changed in the solution process.

We discuss stages -6 to -1 and ≥ 0.

- Stage $k = -6$. We have a scalar equation $f_3 = 0$, which is the first equation in block 4:6; cf. Figure 2.1 and Figure 2.2(c). Since it is nonlinear in x and y, trial values for x and y are necessary, and we need to determine values for them such that $f_3 = 0$ is satisfied.

- Stage $k = -5$. We use the previously computed x, y, give trial values for x', y' and solve $f_3' = 0$, which is linear in x' and y'.

- Stage $k = -4$. We use computed x, x', y, y' to solve $0 = f_1$, f_2, f_3'' for $x''\, y''$, λ; no trial values are needed, as this system is linear in x'', y'', λ.

- Stage $k = -3$. We have two blocks: 4:6 and 1:1. For block 4:6 we use computed x, x', x'', y, y', y'', λ to solve $0 = f_1'$, f_2', f_3''' for x''', y''', λ'. Then the equation in block 1:1 does not require solving, but at this stage, we give an initial value for v, which is used in later stages.

- Stage $k = -2$. We have three blocks: 4:6, 3:3 and 1:1. For block 4:6 we use the computed x, x', x'', x''', y, y', y'', y''', λ, λ' to solve $0 = f_1''$, f_2'', $f_3^{(4)}$ for $x^{(4)}$, $y^{(4)}$, λ''. For block 3:3 we use computed λ, λ', λ'', v and give a trial value for u to solve $0 = f_6$. Finally, for block 1:1 we give an initial value for v' which is used in following stages.

- Stage $k = -1$. Again we have three blocks, 4:6, 3:3 and 1:1. For block 4:6 we use the computed x, x', x'', x''', $x^{(4)}$, y, y', y'', y''', $y^{(4)}$, λ, λ', λ'' to solve $0 = f_1'''$, f_2''', $f_3^{(5)}$ for $x^{(5)}$, $y^{(5)}$, λ'''. For block 3:3 we use the computed λ, λ', λ'', λ''', u, v, v' to solve $0 = f_6'$ for u'. Lastly, for block 1:1 we give an initial value for v'' to be used in the positive number stages.

- Stage $k \geq 0$. We solve linear systems as summarized in Table 2.1.

block	using	solve	for
4:6	$x, x', \ldots, x^{(k+5)}$ $y, y', \ldots, y^{(k+5)}$ $\lambda, \lambda', \ldots, \lambda^{(k+3)}$	$f_1^{(k+4)}, f_2^{(k+4)}, f_3^{(k+6)}$	$x^{(k+6)}, y^{(k+6)}, \lambda^{(k+4)}$
3:3	$\lambda, \lambda', \ldots, \lambda^{(k+4)}$ $u, u', \ldots, u^{(k+1)}$ $v, v', \ldots, v^{(k+2)}$	$f_6^{(k+2)}$	$u^{(k+2)}$
2:2	$u, u', \ldots, u^{(k+2)}$ $\mu, \mu', \ldots, \mu^{(k-1)}$	$f_4^{(k)}$	$\mu^{(k)}$
1:1	$v, v', \ldots, v^{(k+2)}$ $\mu, \mu', \ldots, \mu^{(k)}$	$f_5^{(k)}$	$v^{(k+3)}$

Table 2.1: MOD2PEND: solution process for stages $k \geq 0$.

Compact solution scheme. A more compact representation of the solution scheme is produced by

```
printSolScheme(sadata,'varnames',vars,...
               'outfile','solschemecompact.txt','detail','compact');
```

which results in the file `solschemecompact.txt`:

```
Compact solution scheme for 'modified2pendula' problem
------------------------------------------------------------------------
Initialization summary:
x, x', y, y', u, v, v', v'', v'''
------------------------------------------------------------------------
k = -6: ~[f3] : x, y
k = -5:  [f3'] : x', y'
k = -4:  [f1, f2, f3''] : x'', y'', lam
k = -3:  [f1', f2', f3'''] : x''', y''', lam'
         [] : v
k = -2:  [f1'', f2'', f3''''] : x'''', y'''', lam''
        ~[f6] : u
         [] : v'
k = -1:  [f1''', f2''', f3^(5)] : x^(5), y^(5), lam'''
         [f6'] : u'
         [] : v''
k =  0:  [f1'''', f2'''', f3^(6)] : x^(6), y^(6), lam''''
         [f6''] : u''
         [f4] : mu
        ~[f5] : v'''
```

On the right of ":" is the list of variables for which we solve. The brackets `[...]` mark the system being solved, and `[]` means no equations are solved, but value(s) for variable(s) must be given. The

notation `~[...]` denotes that the block of equations inside these brackets is non-quasilinear in its highest-order derivatives. If no detail level is specified then the compact level is given by default. As in §2.3.3, omitting the key pair `'outfile'` and `'filename.txt'` from the `printSolScheme` function will result in outputs being printed to the command window.

Chapter 3

Theory Overview

In this chapter, we first introduce the basic theory and terminology of structural analysis (SA), §3.1, and then explain the solution method §3.2, the coarse and fine block triangular forms §3.3, and the quasi-linearity analysis (QLA), §3.5.

3.1 Structural analysis

Here we give enough theory that a user can, on small problems, do the analysis by hand. For further details see [3, 4, 9].

Our method gives essentially the same result as does that of Pantelides [7], but is easier to use. The steps to perform SA are as follows.

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

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

2. Find a *highest value transversal* (HVT) of Σ. A *transversal* T is a set of n positions in the matrix with one entry in each row and each column. We denote $\mathrm{Val}(T)$ the sum of the σ_{ij} in those positions, and we aim at finding T such that $\mathrm{Val}(T)$ is as large as possible. The *value of the signature matrix* Σ, written $\mathrm{Val}(\Sigma)$, is defined as the value of any HVT.

   The value of any transversal, and of Σ, is either an integer or $-\infty$. The DAE is *structurally well-posed* (SWP) if $\mathrm{Val}(\Sigma)$ is finite: that is, if there exists at least one transversal all of whose σ_{ij} are finite. Otherwise the DAE is *structurally ill-posed* (SIP), which implies there is some error in the problem formulation.

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

$$d_j - c_i \geq \sigma_{ij} \quad \text{for all } i, j, \tag{3.1}$$

with equality holding on the HVT. They are not unique, but there exist canonical *smallest* offsets, in the sense of $\mathbf{c} \le \mathbf{c}'$ if $c_i \le c_i'$ for all i.

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

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

$$F = \mathrm{Val}(\Sigma) = \sum d_j - \sum c_i.$$

It is the number of *independent* initial values (IVs) it requires, which is the same as the maximum number of IVs that can be specified "fixed".

This is illustrated below for the simple pendulum example, an index-3 DAE which has two HVTs, marked $^\bullet$ and $^\circ$.

Example 1 *Simple pendulum*

$$\begin{aligned} f &= x'' + x\lambda \\ g &= y'' + y\lambda - G \\ h &= x^2 + y^2 - L^2. \end{aligned} \tag{3.2}$$

The signature tableau is

	x	y	λ	c_i
f	$2^\bullet$		0°	0
g		2°	$0^\bullet$	0
h	0°	$0^\bullet$		2
d_j	2	2	0	DOF: 2.

For n up to about 8, setting up Σ and finding a HVT "by eye" is usually easy. Otherwise, let DAESA do the analysis as suggested in §4.2.

4. The $n \times n$ *System Jacobian* matrix is formed as

$$\mathbf{J} = \frac{\partial\left(f_1^{(c_1)}, \ldots, f_n^{(c_n)}\right)}{\partial\left(x_1^{(d_1)}, \ldots, x_n^{(d_n)}\right)}, \qquad \text{or equivalently } \mathbf{J}_{ij} = \begin{cases} \dfrac{\partial f_i}{\partial x_j^{(\sigma_{ij})}} & \text{if } d_j - c_i = \sigma_{ij}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

Equivalence of the equation (3.3) is given in [9]. By f_i' we mean $\mathrm{d}f_i/\mathrm{d}t$, treating the variables and their derivatives as (unknown) functions of t. For instance if $f_1 = x_1'' - x_1 x_3$ then $f_1' = x_1''' - x_1' x_3 - x_1 x_3'$; similarly for higher derivatives. 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, i.e. a point that satisfies all the equations $f_i = 0$, where $\mathbf{J}$ is nonsingular, a solution to the DAE exists through that point; it is locally analytic.

DAESA can permute the system Jacobian $\mathbf{J}$ to upper triangularized block form and identify the structural non-zeros in it.

For the pendulum, (3.3) gives the system Jacobian

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

This is not identically singular since $\det \mathbf{J} = -2(x^2 + y^2)$. Indeed at a consistent point, from the third equation of (3.2), $\det \mathbf{J} = -2L^2 \neq 0$.

5. An upper bound for the standard *differentiation index* ν_d is given by the *structural index*

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

For the commonest kinds of DAE, $\nu_S = \nu_d$, but the difference can be arbitrarily large, [10].

3.2 The solution method

After finding the offsets and HVT for the DAE as described in §3.1, the SA method solves the system in stages numbered as k with

$$f_i^{(k+c_i)} \quad \text{for all } i \text{ such that } k + c_i \geq 0 \tag{3.4}$$

to solve for variables

$$x_j^{(k+d_j)} \quad \text{for all } j \text{ such that } k + d_j \geq 0. \tag{3.5}$$

We start at a negative valued stage equal to $-\max_j d_j$, and at stage 0, we are solving an $n \times n$ system. Thus at any stage $k < 0$, we are using derivatives found either at stage k or at a previous stage, due to the incremental nature of the formula. Use the notation $\mathbf{J}_k$ to denote the system Jacobian at stage k. Then $\mathbf{J}_k$ is a submatrix of $\mathbf{J}$ in the sense of being obtained by selecting certain rows and columns of $\mathbf{J}$, not necessarily contiguous. However, see [9], if we order the equations and variables in descending order of offsets, $\mathbf{J}_k$ becomes the leading $m_k \times n_k$ submatrix of $\mathbf{J}$ for some m_k and n_k, such that $m_k \leq n_k$. The $\mathbf{J}_k$ are nested, in that both m_k and n_k are non-decreasing with k, and equal n when $k \geq 0$.

For the simple pendulum, the system Jacobian takes the needed form when the equations are ordered h, f, g, as shown below. $\mathbf{J}_{-2}$ and $\mathbf{J}_{-1}$, which are equal, are shown boxed.

$$\mathbf{J} = \left[\begin{array}{cc|c} h_x & h_y & 0 \\ \hline f_{x''} & 0 & f_\lambda \\ 0 & g_{y''} & g_\lambda \end{array}\right].$$

The solution scheme for the simple pendulum is given in Table 3.1

k	solve equations	for variables
-2	h	x, y
-1	h'	x', y'
0	f, g, h''	x'', y'', λ
1	f', g', h'''	x''', y''', λ'
$\dots$	$\dots$	$\dots$

Table 3.1: Solution stages for the simple pendulum

The structural analysis method uses (3.4, 3.5) to find Taylor coefficients to solve the DAE via Taylor series, [3].

3.3 The coarse and fine block-triangularizations

A block-triangular form (BTF) of 1.1 is obtained from a *sparsity pattern*, which is some subset A of $\{1, \dots, n\}^2$, the $n \times n$ matrix positions (i, j) for i and j from 1 to n. When appropriate, we identify A with its $n \times n$ *incidence matrix*

$$(a_{ij}) \text{ where } a_{ij} = 1 \text{ if } (i, j) \in A, \text{ 0 otherwise.}$$

A natural sparsity pattern for the DAE is the set where the entries of Σ are finite:

$$S = \{(i, j) \mid \sigma_{ij} > -\infty\} \qquad \text{(the } \textit{sparsity pattern of } \Sigma\text{)}. \tag{3.6}$$

However, a more informative BTF comes from the sparsity pattern of the Jacobian $\mathbf{J}$. It depends, as does $\mathbf{J}$, on the (valid) offset vectors c, d used:

(3.7)

$$S_0 = S_0(c, d) = \{(i, j) \mid d_j - c_i = \sigma_{ij}\} \qquad \text{(the } \textit{sparsity pattern of } \mathbf{J}\text{)}. \tag{3.8}$$

Since $d_j - c_i = \sigma_{ij}$ holds on each HVT by (3.1), and implies $\sigma_{ij} > -\infty$, we have

$$S_0(c, d) \subseteq S \qquad \text{for any } c, d.$$

Experience suggests that in applications, a BTF based on S_0 is usually significantly finer than one based on S. We refer to the former as a *fine* BTF, and we refer to the latter as *coarse* BTF; for more detail see [8].

3.4 Advantage of block-triangularization

We now discuss, by means of an example, why it is advantageous to find a BTF. First, we note that at each k stage in the solution scheme, it may be necessary to prescribe initial values, see §2.3.5.

The equations (3.4, 3.5) can do without initial values, when k is large enough that they become square linear systems. The latest this happens is at stage $k = 0$. However, it can happen for *parts* of the DAE before one reaches stage $k = 0$, as illustrated by (3.9), comprising two simple pendula with a coupling term:

$$\begin{aligned} 0 = A &= x'' + x\lambda, \\ 0 = B &= y'' + y\lambda - G, \\ 0 = C &= x^2 + y^2 - L^2, \\ 0 = D &= u'' + u\mu, \\ 0 = E &= v'' + v\mu - G, \\ 0 = F &= u^2 + v^2 - (L + cx')^2, \end{aligned} \tag{3.9}$$

where G, L, c are constants. Its signature matrix and system Jacobian are

$$\Sigma = \begin{array}{c|cccccc|c} & x & y & \lambda & u & v & \mu & c_i \\ A & 2 & & 0^\bullet & & & & 1 \\ B & & 2^\bullet & 0 & & & & 1 \\ C & 0^\bullet & 0 & & & & & 3 \\ \hline D & & & & 2 & & 0^\bullet & 0 \\ E & & & & & 2^\bullet & 0 & 0 \\ F & 1 & & & 0^\bullet & 0 & & 2 \\ \hline d_j & 3 & 3 & 1 & 2 & 2 & 0 & \end{array}, \qquad \mathbf{J} = \begin{array}{c|ccc|ccc} & x & y & \lambda & u & v & \mu \\ A & 1 & & x & & & \\ B & & 1 & y & & & \\ C & 2x & 2y & & & & \\ \hline D & & & & 1 & & u \\ E & & & & & 1 & v \\ F & \xi & & & 2u & 2v & \end{array}.$$

where $\xi = -2c(L+cx')$

(A blank in Σ means $-\infty$, and in $\mathbf{J}$ means zero.)

One can see just from Σ (or from $\mathbf{J}$), without studying the equations themselves, that the system splits into parts, i.e., subsystems: part 1 has equations A, B, C for variables x, y, λ; part 2 has equations D, E, F for variables u, v, μ. Part 1 influences part 2 by the F, x entry in the lower left block of Σ, but is uninfluenced by it since the top right block is blank, thus giving a block lower-triangular form (BTF).

This coupling leaves the offsets of part 2 unchanged, but increases by one those of part 1—from $c_i = 0, 0, 2,\ d_j = 2, 2, 0$ to $c_i = 1, 1, 3,\ d_j = 3, 3, 1$. This seems to change the initial values part 1 requires—paradoxical, since part 1 is the "uninfluenced" one. Namely, the combined DAE is quasilinear, and by considering the stages defined by (3.4, 3.5), we find that IVs are needed for

$$(x, x', x''; \ y, y', y''; \ \lambda; u, u'; \ v, v').$$

So part 1 now seems to need IVs for x'', y'' and λ, which as a stand-alone system it did not, see Table 3.1. Of course this is false.

When one considers the offsets as defining a scheme for computing successive derivatives (equivalently, Taylor coefficients), the reason for the raised offsets of pendulum 1 is clear. In the absence of coupling, the natural scheme is: at stage $k = -2$, find x, y and u, v; at $k = -1$, find x', y' and u', v'; at $k = 0$, find x'', y'', λ and u'', v'', μ; and so on. However, u, v must satisfy $F = 0$. With the coupling, F involves x', which in the above scheme has not been found yet. Similarly, u', v' must satisfy $F' = 0$, which involves x'', which has not been found yet, and so on. This is cured by shifting pendulum 1 back one stage, so that the scheme becomes:

k	find	*then* find
-3	x, y	
-2	x', y'	u, v
-1	x'', y'', λ	u', v'
0	x''', y''', λ'	u'', v'', μ

and so on. Hence, provided pendulum 1's equations are solved before those of pendulum 2 at each k-stage,

- pendulums 1 and 2 can be solved as separate size 3 systems;
- but, each derivative of x is available just when needed by pendulum 2.

Because of the shift, pendulum 1 behaves as if the overall stages $k = -3, -2, -1, \ldots$ are its *local stages* $\widehat{k} = k + 1 = -2, -1, 0, \ldots$ associated with *local offsets* $\widehat{c}_i = 0, 0, 2$ and $\widehat{d}_j = 2, 2, 0$, which come from analyzing it as a stand-alone system. This shows that the relation between the "minimal initial values" problem and the sequencing of a solution scheme involves the DAE's block lower triangular structure. Hence, we can take advantage of a DAEs block-triangularization by considering local offsets and local solution stages in each block to reduce the number of needed initial values.

3.5 Quasi-linearity analysis

This section illustrates how to do linearity analysis for the blocks we obtain by doing the decomposition.

After permuting the DAE to upper triangularized form of b blocks (where possible), we calculate the *local offsets* $\widehat{c}_i$ and $\widehat{d}_j$ for each block. Then, we analyze the linearity of the highest order derivatives (HODs) of the variables in each block. Denote α_k $(k = 1, \ldots, b)$ such that $\alpha_k = 1$ if the HODs within block i are linear, and $\alpha_k = 0$ otherwise. With the permuted variables, we associate them with an n-vector l such that $l(j)$ is the block of variable x_j $(j = 1, \ldots, n)$. Similarly, we associate the permuted equations with a vector m such that $m(i)$ is the block of equation f_i $(i = 1, \ldots, n)$.

For variables x_j, the required IVs are

$$\mathbf{X}_j = \left(x_j, x_j', \ldots, x_j^{(\widehat{d}_j - \alpha_{l(j)})} \right),$$

where $\widehat{d_j}$ denotes local d offsets here.

For equation f_i in block k, the set of constraints are

$$\mathbf{F}_i = \left(f_i, f_i', \ldots, f_i^{(c_i - \alpha_{m(i)})} \right),$$

where c_i denotes global c offsets.

Therefore, the required IVs for the system are

$$\mathbf{X} = (\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n), \tag{3.10}$$

and the constraints are

$$\mathbf{F} = (\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_n) = \mathbf{0},$$

where some of the $\mathbf{X}_j$'s can be the empty vector, and some of the $\mathbf{F}_i$'s can be missing.

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

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 $\mathbf{X}_j$ in (3.10) 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 $\mathbf{X}_j$ restores local uniqueness.

Chapter 4

DAESA functions

DAESA exploits MATLAB's operator overloading to process the DAE given by a user-supplied function for evaluating the f_i in (1.1). In particular, this is the method DAESA uses to extract the signature matrix and determines for each equation if it is quasilinear in the leading derivatives.

After the signature matrix is constructed, DAESA finds out if the problem is structurally well-posed , and if so, calculates the offsets of the problem and then determines structural index and DOF. Since it knows the structure of the analyzed DAE, DAESA constructs a block-triangular form of it, finds local offsets , and determines block by block quasilinearity. Based on the offsets and linearity information, DAESA deduces which variables and derivatives of them need to be initialized and what the constraints are.

The SA is performed by the function `daeSA`. It returns an object of the class `SAdata`, which encapsulates all the data obtained from the SA. Each of the remaining DAESA functions takes an object of this class as a parameter and extracts from it the data it needs.

In this chapter, we present all the functions in DAESA. In §4.1, we show how a DAE is specified in a function. In §4.2, we introduce the main function, `daeSA`, that performs SA. In §4.3 we describe all the functions and how to obtain structural data from the analysis performed.

4.1 Specifying a DAE

function `f = daefcn(t, x, options)`

`t`
Input: independent variable t

`x`
Input: column n-vector, dependent variables where `x(j)` denotes variable x_j for $j = 1, \ldots, n$

`options`
Input (optional): list of one or more arguments that are passed to the definition of the DAE, useful for parameters

`f`
Output: column n-vector, equations where `f(i)` contains the evaluation of f_i in (1.1)

The code of `daefcn` may contain

- unary operations `+`, `-`
- binary arithmetic operations `+`, `-`, `*`, `/`
- standard functions: **sin**, **cos**, **tan**, **sinh**, **cosh**, **tanh sqrt**, **exp**, **log**, `^`, ...,
  where `^` denotes the "real raised to a real power" function. For a complete list of supported functions, see Appendix A.
- `Dif(var,k)`, the differentiation operator $\mathrm{d}^k/\mathrm{d}t^k$ that returns the `k`th derivative of `var`.
  For instance, `Dif(x(3),2)` represents $x_3'' = \mathrm{d}^2 x_3/\mathrm{d}t^2$.

Notes

- The code should not contain branches, that is `if` or `switch` statements. Functions that are defined using these are not covered by the current theory.
- The code may freely introduce other variables that depend directly or indirectly on the inputs `x` and/or `t`.
- Currently, DAESA supports only scalar operations. Array operations will be implemented in a future version.
- In function `daefcn`, the indices of `x` and `f` are bounded from `1` to `n`. Each variable `x(j)` should be used at least once, and each equation `f(i)` should be evaluated, where `i,j=1,2,...n`.
  As an example of such a function, see Figure 2.1.

4.2 Structural analysis

The structural analysis is performed by the `daeSA` function.

function `sadata = daeSA(daefcn, n, options)`

`daefcn`
Input: a function handle for evaluating the DAE; see p. 20

`n`
Input: positive integer, problem size

`options`
Input (optional): list of one or more arguments that are passed to the definition of the DAE

`sadata`
Output: an `SAdata` class object

This function performs the structural analysis and encapsulates all the data in a return `SAdata` object `sadata`. These SA data can be accessed by the functions described in §4.3.
`daeSA` can fail in the following cases.

1. Problem size n in `daeSA` is not consistent with the number of variables or/and equations encoded in `daefcn`. This can happen if
   (a) variable or equation of index not in `1:n` is accessed
   (b) a variable is not used, and/or an equation is not evaluated.
2. `daefcn` contains functions or operations that are currently not supported.
3. The DAE system defined by `daefcn` is over- or underdetermined.

As long as all the given variable and equation indices are in `1:n`, the structure of the DAE, even if ill-posed, can be shown by `showStruct`, see p. 27.

4.3 Obtaining structural data

If the analyzed DAE is structurally well-posed, we write SWP; otherwise if it is structurally ill-posed, we write SIP. An `SAdata` class object returned by `daeSA` contains data from which one can extract the following information about the DAE:

1. `swp`, a logical value that indicates if the DAE is structurally well-posed
2. `meqn`, `mvar`, indices of missing equations and variables, respectively
3. `index`, structural index

4. `dof`, number of degrees of freedom
5. `S`, signature matrix
6. `HVT` position of a highest value transversal
7. `iv`, set of variables to be initialized
8. `constr`, set of constraints
9. `c`, `d`, `cl`, `dl`, global and local offsets, respectively
10. `pe`, `pv`, permutation row vectors of `1:n` for the triangularized signature matrix
11. `cb`, `fb`, boundaries of the diagonal blocks of the coarse, fine block-triangularization, respectively
12. `cql`, `fql`, quasi-linearity data of diagonal blocks in the coarse and fine block triangularization, respectively
13. `fhandle`, DAE definition function handle
14. `n`, problem size

Each of the remaining functions take as input the object obtained for `daeSA`, which we name `sadata`.

function `swp = isSWP(sadata)`

`swp`
Output: `1` if SWP and `0` otherwise

function `[meqn,mvar] = getMissingEqnsVars(sadata)`

`meqn, mvar`
Output: the indices of equations (in row vector `meqn`) and indices of variables (in row vector `mvar`) that are missing in the DAE definition.

If no equation is missing, `meqn=[]`, and if no variable is missing, `mvar=[]`.

function `index = getIndex(sadata)`

`index`
Output: structural index if SWP and **NaN** otherwise

function `dof = getDOF(sadata)`

`DOF`
Output: DOF if SWP and **NaN** otherwise

function `S = getSigma(sadata)`

`S`
Output: signature matrix in dense form

function `HVT = getHVT(sadata)`

`HVT`
Output: if SWP, a row n-vector such that `(i,HVT(i))`, `i=1:n`, forms a HVT in the unpermuted signature matrix. For example, for the MOD2PEND problem (§2.1), this function returns the vector $(3, 2, 1, 6, 5, 4)$; cf. the HVT in Figure 2.2(a).

If SIP, `HVT` is a row n-vector of **NaN**'s.

function `[c,d] = getOffsets(sadata)`
function `[c,d,cl,dl] = getOffsets(sadata)`

`c, d, cl, dl`
Output: If SWP, `c` and `d` are row n-vectors containing the global equation and variable offsets, respectively; `cl` and `dl` are row n-vectors containing the local equation and variable offsets, respectively.

If SIP, all output row vectors are `[]`.

Note: The number of output arguments should be either 2 or 4, if not an error message will be printed.

function `iv = getInitData(sadata)`

`iv`

Output: If SWP, `iv` is a non-negative integer row n-vector, where `iv(j)` is the number of derivatives of variable j that need initial values when solving the DAE. As a result, the sum of the elements of `iv` is the total number of IVs needed.

For instance, `iv(2)=3` means IVs for x_2, x_2', x_2'' are necessary.
`iv(4)=0` implies x_4 does not participate in initialization.

If SIP, `iv=[]`.

function `constr = getConstr(sadata)`

`constr`

Output: If SWP, `constr` is a non-negative integer row n-vector such that

$$0 = f_i^{(r)} \text{ for } 0 \le r < \texttt{constr(i)}$$

are constraints. As a result, the sum of the elements of `constr` gives the total number of constraints. If SIP, `constr=[]`.

For instance, `constr(3)=3` implies $0 = f_3, f_3', f_3''$ are constraints.
`constr(1)=0` implies f_1 is not in the set of constraints.

If SIP, `constr=[]`.

function `[pe,pv] = getBTF(sadata)`
function `[pe,pv,cb] = getBTF(sadata)`
function `[pe,pv,cb,fb] = getBTF(sadata)`

Let `S=getSigma(sadata)`.

Assume SWP

`pe, pv`
Output: `pe`, `pv` are two row n-vectors such that `S(pe,pv)` is block-upper triangularized signature matrix, where the entries below the diagonal blocks are all $-\infty$. The ith equation in the permuted system is `pe(i)`, and the jth variable in the system is `pv(j)`.

`cb, fb`
Output: `cb`, `fb` are row integer vectors that specify the boundaries of the diagonal blocks of the coarse and fine block-triangularization, respectively.

If `B=S(pe,pv)`, the (i,j)th block of the coarse block-triangularization is `B(cb(i):cb(i+1)-1,cb(j):cb(j+1)-1)`. Similarly, the (i,j)th block of the fine block-triangularization is `B(fb(i):fb(i+1)-1,fb(j):fb(j+1)-1)`.

Assume SIP

`pe, pv`
Output: `pe`, `pv` are two row n-vectors such that `S(pe,pv)` is block-upper triangularized signature matrix, where the entries below the diagonal blocks are all $-\infty$. The ith equation in the permuted system is `pe(i)`, and the jth variable in the system is `pv(j)`.

`cb`
Output: `cb` an empty array

`fb`
a $2 \times m$ positive integer matrix specifying the boundaries of the diagnostic (see below) block triangularization, where m is the number of fine blocks minus one.

If `B=S(pe,pv)`, the (i,j)th block of the diagnostic block-triangularization is:
`B(fb(1,i):fb(1,i+1)-1, fb(2,j):fb(2,j+1)-1)`

Note: The number of output arguments should be either 2, 3 or 4.

Practically, the diagnostic block-triangularized form will look like:

$$\begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ & & B_{23} & B_{24} \\ & & & B_{34} \\ & & & B_{44} \end{bmatrix}$$

where

- block $[B_{11}\ B_{12}]$ is under-determined,
- block B_{23} is well-determined, and
- block $[B_{34}\,;\, B_{44}]$ is over-determined.

function `cql = getQLdata(sadata)`
function `[cql,fql] = getQLdata(sadata)`

Assume SWP

`cql, fql`
Output: row vectors containing information about the quasi-linearity of each diagonal block in the coarse and fine block-triangularizations of a DAE structure, respectively.

Let `[pe,pv,cb,fb] = getBTF(sadata)`.
For the coarse block triangularization, if `cql(i)=true`, then the ith diagonal block comprising equations of indices `pe(cb(i):cb(i+1)-1)` in variables of indices `pv(cb(i):cb(i+1)-1)` is quasilinear in the leading derivatives of these variables; otherwise, `cql(i)=false`, and this block is non-quasilinear.

Similarly, for the fine block triangularization, if `fql(i)=true`, then the ith diagonal block comprising equations of indices `pe(fb(i):fb(i+1)-1)` in variables of indices `pv(fb(i):fb(i+1)-1)` is quasilinear in the leading derivatives of these variables; otherwise, `fql(i)=false`, and this block is non-quasilinear.

Assume SIP

All output row vectors are `[]`.

function `h = getDAEfhandle(sadata)`

`h`
Output: returns a handle of the function passed to `daeSA`

function `n = getSize(sadata)`

`n`

Assume SWP

Output: returns problem size, the number of equations and variables

Assume SIP

Output: if the problem is missing variables or equations the result is still the same, the function returns the number of variables and equations specified in the initial `daeSA` function call.

4.4 Visualization

function `showStruct(sadata,options)`

Assume SWP

This function displays the structure of the signature matrix in the current figure, or in a new figure if none is open. `showStruct` can have in `options` one to three optional key-value pair arguments explained as follows.

`'disptype',typeValue`

`typeValue` is a string that indicates how the structure the DAE should be displayed.

`typeValue`	displays
`'original'` (default)	original structure of the DAE as given. `showStruct(sadata, 'disptype','original')` is the same as `showStruct(sadata)`.
`'blocks'`	coarse block-triangularized structure of the DAE
`'fineblocks'`	fine block-triangularized structure of the DAE

`'submat',matValue`

`matValue` is a row vector of size 2 or 4 specifying what submatrix of the signature matrix `S` to be displayed.

size	displays
2	diagonal submatrix comprising rows `matValue(1)` to `matValue(2)` and columns `matValue(1)` to `matValue(2)`
4	submatrix comprising rows `matValue(1)` to `matValue(2)` and columns `matValue(3)` to `matValue(4)`

`'blocksubmat',blkValue`

`blkValue` is a row vector of size 2 or 4 specifying what blocks of the signature matrix `S` to be displayed.

size	displays
2	diagonal blocks comprising block rows `blkValue(1)` to `blkValue(2)` and block columns `blkValue(1)` to `blkValue(2)`
4	submatrix comprising block rows `blkValue(1)` to `blkValue(2)` and block columns `blkValue(3)` to `blkValue(4)`

Note: to use the key pair `'blocksubmat'`, `blkValue` the `'disptype'` must be set to give a block triangularization.

Assume SIP

A similar set of outputs to the SWP case are available, except the `'disptype'`, `'fineblocks'` is no longer available and the `'disptype'`, `'blocks'` now produces a diagnostic block-triangularization, as detailed below and in [2].

Contents of figure if SWP. Without using arguments `'submat'` or `'blocksubmat'`, the figure displays the DAE function name, size, structural index and DOF.

- In the default setting, the figure also displays
  - `S` signature matrix
  - indices of equations and variables
  - `c,d` global offsets
  - HVT position (boxed in BW print, yellow in color print)
  - structural non-zeros in the system Jacobian (shaded in BW print, colored green in color print)
- With `'disptype','blocks'` or `'disptype','fineblocks'`, the figure displays:
  - `B` , permuted signature matrix (`B=S(pe,pv)`)
  - `pe, pv`, permuted indices of equations and variables
  - `c, d, cl, dl`, global and local offsets
  - union of HVTs (boxed in BW print, yellow in color print)
  - structural non-zeros in the system Jacobian (shaded in BW print, colored green in color print)
- With argument `'submat',matValue` or `'blocksubmat',blkValue`, the figure only displays:
  - DAE function name
  - submatrix (or blocks) information
  - submatrix (or blocks) of the signature matrix
  - indices of equations and variables
  - union of HVTs (boxed in BW print, yellow in color print)
  - structural non-zeros in the system Jacobian (shaded in BW print, colored green in color print)

If either the width or height of the matrix displayed is larger than 40, the figure does not show the indices and the offsets.

Contents of figure if SIP. If any variable or equation of index in `1:n` is not accessed, the DAE is SIP, and the equations or variables that make the DAE SIP will be highlighted (dark shading in BW print, colored red in color print), and figure will display:

- When no `'disptype'` is specified:
  - DAE function name

  - structurally ill posed
  - indices of variables and equations

- When `'disptype'` is `'blocks'`:
  - DAE function name
  - structurally ill posed
  - indices of variables and equations
  - missing equations/variables
  - a diagnostic BTF, as explained in the description of `getBTF` on p. 25 and shown in Figure 5.1 in §5.2

- When `'disptype'` is `'fineblocks'` the user will receive a message, printed to the command window, telling them the DAE is SIP and fine blocks cannot be displayed:

  ```
  'daefcn' is structurally ill posed. Fine blocks cannot be displayed.
  ```

4.5 Data output

For the following functions, we assume SWP. If SIP an error message is printed to the command window:

```
'daefcn' is structurally ill posed.
```

function `printInitData(sadata, options)`

This function reports the variables (and derivatives of them) that need to be initialized for solving the DAE.

This function can have in `options` one or two key-value pairs, where the keys are `'varnames'` and `'outfile'`.

`'varnames',vars`
By default, the variable names are `x1`, `x2`, ..., `xn`. If each of the variable names needs to be replaced by the same variable name, say `v`, followed by index number, one can pass `'varnames','v'`. Alternatively if one would like to specify each variable.

`'varnames',cavarnames`
Different variable names can be given in a cell array of n strings, say `cavarnames`, and then passed as `'varnames',cavarnames`. This results in the ith variable name being replaced by `cavarnames{i}`.

`'outfile',fname`
By default, the text is printed on the screen. If optional `'outfile','fname'` is passed, the output is stored in a file with name `fname`.

function `printConstr(sadata, options)`

This function reports the functions (and derivatives of them) that form the set of the constraints.

This function can have in `options` one or two key-value pairs, where the keys are `'fcnnames'` and `'outfile'`.

`'fcnnames',fcnname`

`'fcnnames',cafcnnames`

By default, the function names are `f1`, `f2`, ..., `fn`. If function names are different from the default ones, they can be replaced in the output in the same way as in `printInitData`. For instance, `'fcnnames','g'` replaces each of the function names with `g` followed by index number; `'fcnnames',cafcnnames` replaces the default function names with different function names, where `cafcnnames` is a cell array of n strings.

`'outfile',fname`

The same as above in `printInitData`.

function `printSolScheme(sadata, options)`

This function reports a solution scheme for the DAE.
`options` is an optional list of one to four key-value pair arguments from

`'varnames'`,	`varname` or `cavarnames`
`'fcnnames'`,	`fcnname` or `cafcnnames`
`'outfile'`,	`fname`
`'detail'`,	`detaillevel`

The first three pairs are as described in `printInitData` and `printConstr`. The `detaillevel` can be `'compact'` (default), which causes this function to report the solution scheme in compact form, or `'full'`, which results in a more detailed output. See §2.3.5 for more information on the `'detail'` option.

In the three functions above, the output file `fname` can be specified as a full path, e.g. `'c:/My Documents/My_DAESA_outputs/my_files_name.txt'`. If no full path is specified then the path is relative to the current MATLAB working directory. For example, `'Reports/filename.txt'` will print to a `Reports` folder in the current directory. For all the above print functions the format is plain text.

Data output samples using these functions can be found in §2.3.

Chapter 5

Examples

We now work through examples of some of the functions of the previous chapter. We start with the SWP `modified2pendula` example (§5.1), and then we show how DAESA visualizes problems that are structurally ill-posed (§5.2). Then we present results from applying DAESA on the chemical Akzo Nobel problem [1] (§5.3) and a problem consisting of several coupled pendula (§5.4). In all following examples, it will be assumed that `sadata` refers to the result of calling `daeSA` on the `daefcn` being explained.

5.1 Well-posed DAE example

First, we use the SWP case example. Recall the result of `showStruct` on the `modified2pendula` example, displayed in Figure 2.2. We now apply the functions `getHVT` and `getOffsets` to this example. Returning

```
>> HVT=getHVT(sadata)
HVT =
     3     2     1     6     5     4
>> [c,d,cl,dl]=getOffsets(sadata)
c =
     4     4     6     0     0     2
d =
     6     6     4     2     3     0
cl =
     0     0     2     0     0     0
dl =
     2     2     0     0     3     0
```

which when compared to the results of Figure 2.2 should give the reader a better understanding of these functions. The local offsets are returned in the original equation order and not in the permuted block triangular equation order shown by calling `showStruct` with `'disptype'` set as `'fineblocks'`. If we wanted to change the ordering, the commands using `getBTF` detailed below should be used.

We may have a large solution scheme produced from the `printInitData` and `printConstr` functions of §(2.3.3) and as such may want a more compact representation of the data. To do this we

use `getInitData` and `getConstr` to produce

```
>> iv=getInitData(sadata)
iv =
     2     2     0     1     4     0
>> constr=getConstr(sadata)
constr =
     4     4     6     0     1     3
```

so, for example variable one and its derivative are needed initial conditions for the solution process, as already shown with `printInitData` in §2.3.3.

Finally, we consider the result of applying `getBTF` and `getQLdata` to the `modified2pendula` example

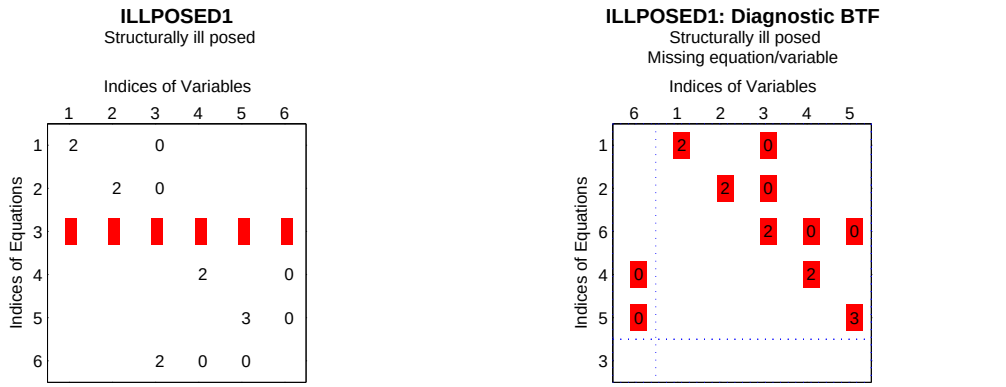
```
>> [pe,pv,cb,fb]=getBTF(sadata)
pe =
     5     4     6     3     2     1
pv =
     5     6     4     1     2     3
cb =
     1     4     7
fb =
     1     2     3     4     7
>> [cql,fql] = getQLdata(sadata)
cql =
     0     1
fql =
     0     1     0     1
```

Recalling Figure 2.2, in particular the blocks in the coarse and fine cases, should give the user a good understanding of the outputs of these functions. Finally, if we wanted to produce now the local offsets in the BTF ordering given by calling `showStruct` with `'disptype'` set as `'fineblocks'` then we would use the following commands:

```
>> cl=cl(pe)
cl =
     0     0     0     2     0     0
>> dl=dl(pv)
dl =
     3     0     0     2     2     0
```

5.2 Ill-posed DAE examples

We now turn our attention to the SIP problem resulting from removing equation three in the `modified2pendula` example mentioned above. By applying `showStruct` we obtain the following figures:



(a) original structure (b) diagnostic block-triangularized structure

Figure 5.1: Structure of `illPosed1` and its diagnostic block-triangularization.

If we apply `getBTF` to this example we obtain:

```
>> [pe,pv,cb,fb]=getBTF(sadata)
pe =
     1     2     6     4     5     3
pv =
     6     1     2     3     4     5
cb =
     []
fb =
     1     6     6     6     7
     1     2     7     7     7
```

see also p. 25 .

Now, to show how DAESA displays missing equations and variables, we consider `illPosed2`, which is the `modified2pendula` example omitting both equation three and variable μ. A call of `showStruct` yields Figure 5.2.

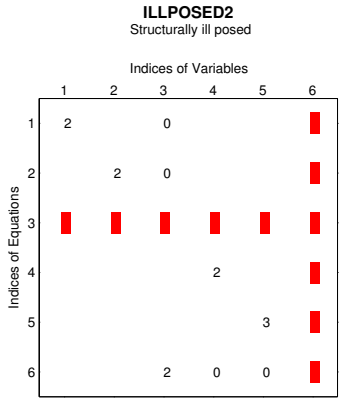


Figure 5.2: Structure of `illPosed2`.

In this example the missing equations and variables are made obvious by the shaded entries (red in color print). However for a larger example, we may not be able to easily distinguish missing equations and variables. A call of `getMissingEqnsVars` yields:

```
>> [meqn,mvar]=getMissingEqnsVars(sadata)
meqn =
     3
mvar =
     6
```

Finally, we show how DAESA displays several diagnostic blocks for a SIP DAE. We use `illPosed3` from the examples directory, see Figure 5.3

```
function f = illPosed3(t, z)
 x1 = z(1); x2 = z(2); x3 = z(3);
 x4 = z(4); x5 = z(5); x6 = z(6);
 f(1) = x1    + x2 +            x5    + x6   ;
 f(2) = x1^3 + x2 +             x5    + x6   ;
 f(3) =                         x5    * x6   ;
 f(4) =                       - x5^3  + x6^4;
 f(5) = x1    + x2 + x3 + x4 + x5    + x6^3;
 f(6) =                         x5    + x6   ;
end
```

Figure 5.3: MATLAB function for evaluating `illPosed3` example.

Calling `showStruct` with `'disptype'` set as `'blocks'` yields the diagnostic block structure seen in Figure 5.4.

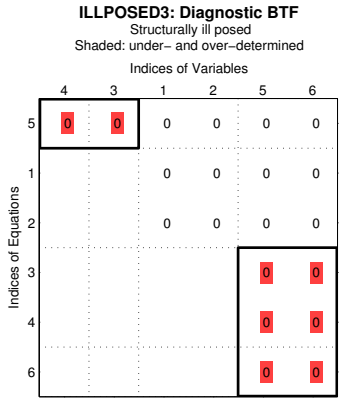


Figure 5.4: Diagnostic block-triangularized structure of `illPosed3`.

Here, there is a structurally under-determined set of variables x_4 and x_3 in equation $0 = f_5$, a structurally over-determined set of equations $0 = f_3, f_4, f_6$ in variables x_5 and x_6 and a structurally well-determined system in variables x_1 and x_2 in equations $0 = f_1, f_2$. For more information on the diagnostic block-triangularization see the description of `getBTF` on p. 25.

5.3 Chemical Akzo Nobel

We show DAESA's SA on the chemical Akzo Nobel problem [1], an index-1 DAE. The function for evaluating the corresponding equations is in Figure 5.5.

```
function f = akzonobel(t,y)
 k1 = 18.7; k2 = 0.58;   k3 = 0.09; k4 = 0.42;
 K  = 34.4; klA = 3.3; CO2 = 0.9;    H = 737;
 Ks = 115.83;
 r1  = k1*y(1)^4*sqrt(y(2));
 r2  = k2*y(3)*y(4);
 r3  = k2/K*y(1)*y(5);
 r4  = k3*y(1)*y(4)^2;
 r5  = k4*y(6)^2*sqrt(y(2));
 Fin = klA*(CO2/H - y(2));
 f(1) = -Dif(y(1),1) - 2.0*r1 + r2 - r3 - r4;
 f(2) = -Dif(y(2),1) - 0.5*r1 - r4 - 0.5*r5 + Fin;
 f(3) = -Dif(y(3),1) + r1 - r2 + r3;
 f(4) = -Dif(y(4),1) - r2 + r3 - 2.0*r4;
 f(5) = -Dif(y(5),1) + r2 - r3 + r5;
 f(6) = Ks*y(1)*y(4) - y(6);
end
```

Figure 5.5: DAESA function for evaluating the chemical Akzo Nobel DAE.

The script in Figure 5.6 produces the solution scheme in Figure 5.7 and the plot in Figure 5.8.

```
sadata = daeSA(@akzonobel,6);
showStruct(sadata,'disptype','fineblocks');
printSolScheme(sadata,'outfile','chemakzo.txt','varnames','y');
```

Figure 5.6: DAESA script for analyzing `AkzoNobel`.

```
Solution scheme for 'akzonobel' problem
-------------------------------------------------------------------------
Initialization summary:
y1, y2, y3, y4, y5
-------------------------------------------------------------------------
k = -1:  [] : y5
         [] : y4
         [] : y3
         [] : y2
         [] : y1
k =  0:  [f6] : y6
         [f5] : y5'
         [f4] : y4'
         [f3] : y3'
         [f2] : y2'
         [f1] : y1'
```

Figure 5.7: Akzo Nobel: solution scheme.

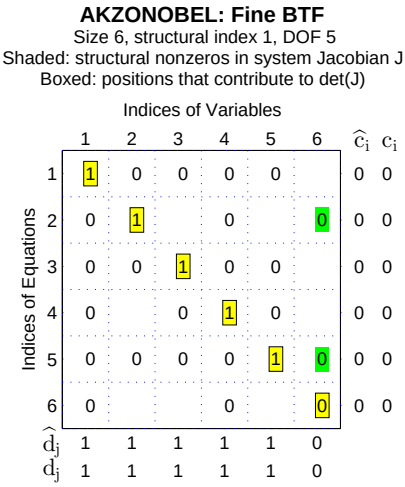


Figure 5.8: Fine block-triangularized structure of the chemical Akzo Nobel problem.

This scheme suggests that, at stage $k = -1$, we should give initial values for y_i for all $i = 1, \ldots, 5$. For stages $k \geq 0$, we proceed as shown in Table 5.1.

block	solve	for
6:6	$f_6^{(k)}$	$y_6^{(k)}$
5:5	$f_5^{(k)}$	$y_5^{(k+1)}$
4:4	$f_4^{(k)}$	$y_4^{(k+1)}$
3:3	$f_3^{(k)}$	$y_3^{(k+1)}$
2:2	$f_2^{(k)}$	$y_2^{(k+1)}$
1:1	$f_1^{(k)}$	$y_1^{(k+1)}$

Table 5.1: Solution scheme for the Akzo Nobel problem for $k \geq 0$.

In [1], this problem is classified as nonlinear, and initial values for all variables and their first derivatives are given. Our solution scheme reveals six linear equations, if solved in a block-wise manner, and reports that only $y_1, \ldots, y_5$ need to be initialized.

5.4 Multiple pendula

In this example, we illustrate using a **for** loop in the function for evaluating a DAE system, where the number of iterations is passed as a parameter. Consider the "chain" of pendula:

$$\begin{array}{ll} \text{first pendulum} & i\text{th pendulum} \\ 0 = x_1'' + \lambda_1 x_1 & 0 = x_i'' + \lambda_i x_i \\ 0 = y_1'' + \lambda_1 y_1 - G & 0 = y_i'' + \lambda_i y_i - G \\ 0 = x_1^2 + y_1^2 - L^2 & 0 = x_i^2 + y_i^2 - (L + c\lambda_{i-1})^2, \end{array} \tag{5.1}$$

where the state variables of the ith pendulum ($i \geq 1$) are x_i, y_i, and λ_i; $G > 0$ is gravity, $L > 0$ is the length of the first pendulum, and c is a constant. For $i \geq 2$, the length of the ith pendulum depends on λ_{i-1}. A system of p pendula has size $n = 3p$ and index $2p + 1$, [2, 5].

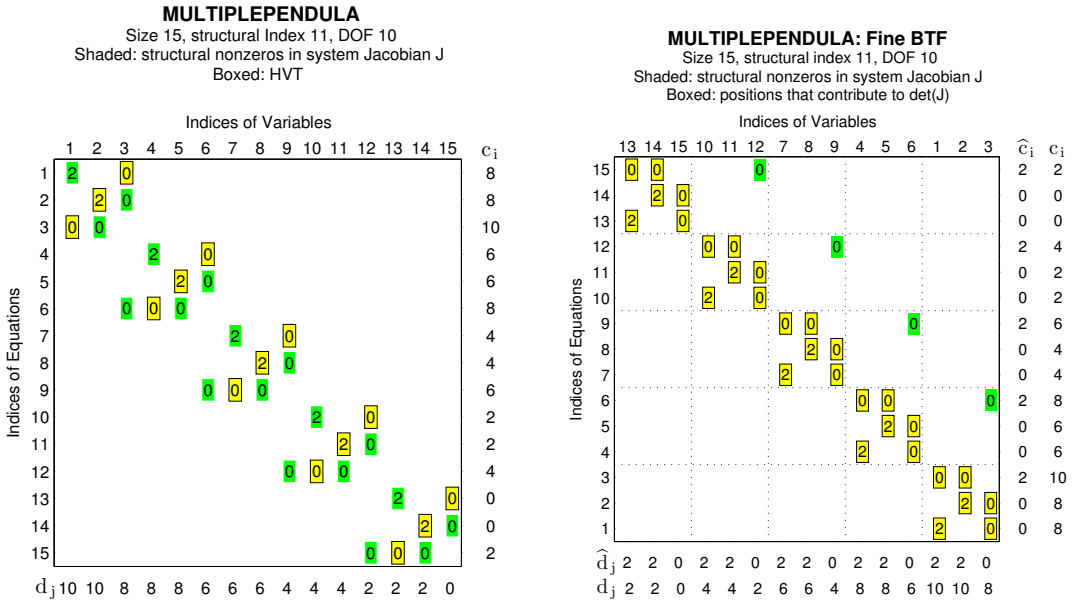
The function for evaluating (5.1) is in Figure 5.9. Here, the number of pendula, `p`, is passed as a parameter. In the **for** loop, we evaluate the equations for pendulum $i = 2, \ldots, p$. The `daeSA` function can be called as

```
p = 5; n = 3*p;
sadata = daeSA(@multiplependula,n,p);
```

In Figure 5.10, we show the original and fine block-triangularizations of this problem (the coarse and fine triangularizations are the same).

```
function f = multiplependula(t,x,p)
 G = 9.8; L = 1; c = 0.1;
 % first pendulum
 f(1) = Dif(x(1),2) + x(1)*x(3);                    % 0 = x_1'' + λ_1 x_1
 f(2) = Dif(x(2),2) + x(2)*x(3)-G;                  % 0 = y_1'' + λ_1 y_1
 f(3) = x(1)^2 + x(2)^2 - L^2;                      % 0 = x_1^2 + y_1^2 - L^2
 % pendulum > 1
 for i = 2:p
     xi = 3*i-2; yi = 3*i-1;                        % x, y indices
     li = 3*i;  li1 = 3*i-3;                        % λ_i, λ_{i-1} indices
     f(xi) = Dif(x(xi),2) + x(xi)*x(li);            % 0 = x_i'' + λ_i x_i
     f(yi) = Dif(x(yi),2) + x(yi)*x(li) - G;        % 0 = y_i'' + λ_i y_i - G
     f(li) = x(xi)^2 + x(yi)^2 - (L+c*x(li1))^2;
                                                    % 0 = x_i^2 + y_i^2 - (L + cλ_{i-1})^2
 end
end
```

Figure 5.9: DAESA function for evaluating the multiple pendula problem.



(a) original structure

(b) fine block-triangularized structure

Figure 5.10: Structure of (5.1) and its fine block-triangularization.

Chapter 6

Installation

This package is available at `http://www.cas.mcmaster.ca/~nedialk/daesa` as the zipped file `DAESA-1.0.zip`. The implementation is distributed in the form of MATLAB `pcode` files.

When `DAESA-1.0.zip` is unzipped, it creates directory `DAESA-1.0` in the current directory with subdirectories as follows:

subdirectory	contains
`src/implementation`	`pcode` files of the implementation
`src/interface`	`.m` files that execute the corresponding `pcode` files
`examples`	the code for the examples in this User Guide
`examples/DAEs`	the functions for evaluating the DAEs used in this User Guide

Before using DAESA, one should set paths to the above directories. This can be done by executing **startup**`.m` in directory `DAESA`, or by setting them manually through MATLAB's GUI.

Example programs. The results in this user guide are produced by the following scripts in the `examples` directory:

file	used in	DAE function in `examples/DAEs`
`sa_modified2pendula.m`	§5.1	`modified2pendula.m`
`sa_illPosed.m`	§5.2	`illPosed1.m`, `illPosed2.m`, `illPosed3.m`
`sa_chemakzo.m`	§5.3	`akzonobel.m`
`sa_multiplependula.m`	§5.4	`multiplependula.m`

(The remaining files in the `examples` and `examples/DAEs` directories are used in [2].)

Once DAESA is on MATLAB's path it can be used within MATLAB from an arbitrary working directory. If this is done, whenever an `'outfile'` is specified in the **print** functions of §2.3.3, the file path (if not an absolute path) will be relative to the current working directory.

Appendix A

Supported standard functions

The following two tables list all standard functions currently supported by DAESA.

DAESA name	resulting function
^	power, where the power is a real number
exp	exponential
log	natural logarithm
log10	base 10 logarithm
sqrt	square root
sin	sine
cos	cosine
tan	tangent
sec	secant
csc	cosecant
cot	cotangent
asin	arcsine
acos	arccosine
atan	arctangent
asec	arcsecant
acsc	arccosecant
acot	arccotangent

DAESA name	resulting function
sinh	hyperbolic sine
cosh	hyperbolic cosine
tanh	hyperbolic tangent
sech	hyperbolic secant
csch	hyperbolic cosecant
coth	hyperbolic cotangent
asinh	hyperbolic arcsine
acosh	hyperbolic arccosine
atanh	hyperbolic arctangent
asech	hyperbolic arcsecant
acsch	hyperbolic arccosecant
acoth	hyperbolic arccotangent

Table A.1: Standard functions supported by DAESA.

Index

Bibliography

[1] 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/`.

[2] N. S. Nedialkov, J. Pryce, and G. Tan, *DAESA — a Matlab tool for structural analysis of DAEs: Software*, 2013. Accepted for publication in ACM TOMS.

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

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

[5] ——, *Solving differential-algebraic equations by Taylor series (III): The DAETS code*, JNAIAM, 3 (2008), pp. 61–80. ISSN 1790ÃŘ8140.

[6] ——, *DAETS user guide*, tech. rep., Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, L8S 4K1, 2008–2009.

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

[8] J. Pryce, N. S. Nedialkov, and G. Tan, *DAESA — a Matlab tool for structural analysis of DAEs: Theory*, 2013. Accepted for publication in ACM TOMS.

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

[10] 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 (1999), pp. 1987–1990.