Technische Universität Berlin
Institut für Mathematik

A MATLAB Toolbox for the Numerical Solution of Differential-Algebraic Equations

Peter Kunkel, Volker Mehrmann, and Stefan Seidel

Preprint 16/2005

Preprint-Reihe des Instituts für Mathematik
Technische Universität Berlin

Report 16/2005 July 2005

A MATLAB Toolbox for the Numerical Solution of Differential-Algebraic Equations

Peter Kunkel [*] Volker Mehrmann [†] Stefan Seidel [‡]

July 21, 2005

http://www.synoptio.de/Math_Contact.html
http://www.math.tu-berlin.de/~mehrmann

[*] Mathematisches Institut, Universität Leipzig, Augustusplatz 10–11, D-04109 Leipzig, Fed. Rep. Germany.

[†] Institut für Mathematik, MA 4-5, Technische Universität Berlin, Straße des 17. Juni 136, D-10623 Berlin, Fed. Rep. Germany.

[‡] SynOptio GmbH, Torstraße 23, D-10119 Berlin, Fed. Rep. Germany.

Contents

1 Introduction

1.1 Purpose of this Toolbox

The purpose of this MATLAB [1] toolbox is the numerical solution of systems of linear and nonlinear differential-algebraic equations (DAEs).

This first release of the toolbox treats systems of linear DAEs of the form

$$E(t)\dot{x}(t) = A(t)x(t) + f(t),\ x(t_0) = x_0 \tag{1}$$

where $t \in [t_0, t_f],\ x : \mathbb{R} \to \mathbb{R}^n, f : \mathbb{R} \to \mathbb{R}^l$ and $E, A : \mathbb{R} \to \mathbb{R}^{l\times n}$, and square nonlinear DAEs of the form

$$F(t, x(t), \dot{x}(t)) = 0,\ x(t_0) = x_0 \tag{2}$$

where $t \in [t_0,\ t_f],\ x : \mathbb{R} \to \mathbb{R}^n$ and $F : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$.

In order to solve these types of DAEs, the Fortran subroutines **GELDA** (**GE**neral **L**inear **D**ifferential-**A**lgebraic equation solver) [7] and **GENDA** (**GE**neral **N**onlinear **D**ifferential-**A**lgebraic equation solver) [8] have been developed. The toolbox employs these two solvers by providing MEX-files that turn these Fortran subroutines into MATLAB callable functions. However, the toolbox is designed to make the utilization of **GELDA** and **GENDA** as easy and comfortable as possible, so that the user does not have to dig deeply into the details of the Fortran code. Instead, all the setup and customization can be done using a graphical user interface (GUI). Furthermore, this toolbox features symbolic differentiation. In this way, one only has to supply the functions defining the given DAE and one does not need to provide the derivatives or Jacobians. Other special features of this toolbox include

- the computation of characteristic values,
- the computation of consistent initial values in the least square sense,
- many parameters that enable the user to customize the solvers to certain problems.

For questions concerning licensing of the MATLAB DAE Toolbox contact SynOptio GmbH (visit `http://www.synoptio.de/Math_Contact.html`).

1.2 Requirements

The toolbox has been written for MATLAB Release 14 Service Pack 2. To take advantage of all the features of it you need the following MATLAB toolboxes.

- Optimization Toolbox
- Symbolic Math Toolbox

[1] MATLAB© is a registered trademark of The MathWorks, Inc. To simplify the presentation this brand name will be used without the copyright symbol throughout this document.

It is possible to run the DAE toolbox without the toolboxes mentioned above. If the Optimization Toolbox is not available, one cannot try to correct guesses of consistent initial values (see section 2). One cannot use the symbolic differentiation feature if the Symbolic Math Toolbox is not installed (see sections 2 and 5).

2 Solving DAEs

This section covers the basic knowledge that is needed to solve a certain DAE using the toolbox. Typically, the procedures shown in Figure 1 are used.

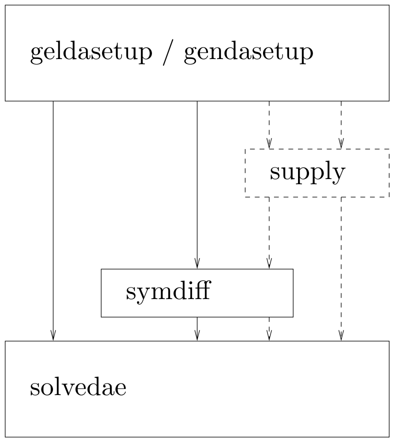

Figure 1: Ways to solve a DAE using this toolbox

The first step to compute a solution for a given problem is to call the function **geldasetup**, if a linear DAE is considered, or **gendasetup** if a nonlinear DAE is considered. These functions provide means to edit the setup of the solver being used via a GUI. Their output is a structure array which contains all information that is necessary to run a solver. Unfortunately, the input of vectors and other sophisticated data types, such as function handles, via the GUI is not possible. If one wants to provide one of these, the function **supply** is called which saves these data types to the structure array. The next step is to provide the necessary derivatives. There are two possibilities. The first one is to use the symbolic differentiation feature of this toolbox. This means that one has to call the function **symdiff** (see section 5 for more details). The second possibility is to use user supplied derivatives (see section 6 for further information). No matter how one obtains the derivatives, the final step is to call the function **solvedae**. Depending on the information given during the setup phase this function picks the right solver for the considered problem. There are two solvers available. Their names and their area of application is shown in Table 1.

lindaesolve	solves linear DAEs using **GELDA**
nldaesolve	solves nonlinear DAEs using **GENDA**

Table 1: Solvers and their areas of application

The function **solvedae** exists for the sake of operator convenience. Of course, it is possible to call the solvers listed in Table 1 directly but the advantage of **solvedae** is that the user has to memorize only one function name.

With these proceedings in mind, we take a look at the overview of all functions being part of this toolbox. The following list is not complete. There are many more functions making the toolbox work (see appendix A). However, it is very unlikely that situations will arise, where the user is forced to use one of these directly.

- ## geldasetup

  initializes a structure array that contains setup information for the linear DAE solver

  ### Syntax

  `s = geldasetup(m, n, tspan)`

  `s = geldasetup(m, n, tspan, setting)`

  ### Arguments

  The following table describes the input arguments to the function geldasetup.

m	number of equations
n	number of components of the solution vector x
tspan	1×2 array specifying t_0 and t_f
setting	string specifying the setting which is loaded 'standard': standard setup which is also loaded as default if *setting* is omitted 'large': setup for large systems

  The following table lists the output arguments for the function geldasetup.

s	structure array containing all information about the setup of the solver

  ### Description

  `s = geldasetup(m, n, tspan, [setting])` opens a GUI where the user can edit the setup for the solver **lindaesolve**. Necessary information about the setup is stored in the structure array `s`.

- ## gendasetup

  initializes a structure array containing setup information for the nonlinear DAE solver

Syntax

```
s = gendasetup(n, tspan, cval)
s = gendasetup(n, tspan, cval, setting)
```

Arguments

The following table describes the input arguments to the function gendasetup.

n	number of equations
tspan	1×2 array containing t_0 and t_f
cval	4×1 array containing the characteristic values of the DAE (2) in the following order $cval(1)$ = strangeness-index (see section 9) $cval(2)$ = number of differential components $cval(3)$ = number of algebraic components $cval(4)$ = number of undetermined components
setting	string specifying the setup which is loaded 'standard': standard setup which will be also loaded as default if *setting* is omitted 'multibody': default setup for multi-body systems

The following table lists the output arguments for the function gendasetup.

s	structure array containing all information about the setup of the solver

Description

`s = gendasetup(n, tspan, cval, [setting])` opens a GUI where the parameters influencing the solver **nldaesolve** can be edited. The user has to specify the number of equations, and the starting point and the end point of integration as well as the characteristic values of the DAE. The result of this function is a structure array which contains all information about the setup of the solver that are necessary to run it.

- ## supply

saves vectors and function handles in an existing structure array

Syntax

```
s = supply(what, vec, ...)
s = supply(what, handle,...)
```

Arguments

The following table describes the input arguments to the function supply.

what	string specifying the kind and purpose of the next argument possibilities are vectors and function handles 'ifix': vector prescribing differential variables (see section 4.2) 'scalc': column scaling vector (see section 4.2) 'scalr': row scaling vector (see section 4.2) 'rtol': relative error tolerance as vector 'atol': absolute error tolerance as vector 'uscal': handle to a function for scaling purposes
vec	vector which will be saved in the field of structure array *s* specified by *what*
handle	handle to function for scaling purposes only applicable if *what* = 'uscal'

The following table lists the output arguments for the function supply.

s	structure array containing all information about the setup of the solver

Description

`s = supply(what, vec, what, handle, ...)` saves vectors or function handles in an existing structure array. If, for example, the user wants to provide error tolerances as vectors this function must be used. The input must be given as pairs consisting of a string input specifying which vector is given and the vector itself. All inputs are optional and can be given in any order. If the user enters RTOL, ATOL must be supplied also and vice versa. The user can also supply a handle to a function which is used for scaling purposes (called from the subroutine uscal needed for **GENDA**).

For **GELDA** supply can only be used to enter RTOL and ATOL.

- **symdiff**

  symbolically differentiates functions

  ### Syntax

  `[edif, adif, fdif] = symdiff(s, eh, ah, fh, name1, name2, name3)`

  `[fdif, dfdif]  = symdiff(s, fh, name1, name2)`

  `[fdif, dfdif, x0]  = symdiff(s, fh, name1, name2, giv)`

  ### Arguments

  The following table describes the input arguments to the function symdiff.

s	structure array containing all information which are necessary to setup this solver
linear case	
eh	handle to a function specifying E
ah	handle to a function specifying A
fh	handle to a function specifying f
name1	name for the function implementing E and its derivatives
name2	name for the function implementing A and its derivatives
name3	name for the function implementing f and its derivatives
nonlinear case	
fh	handle to a function specifying F
name1	name for the function implementing F and its derivatives
name2	name for the function implementing the Jacobians of $F, \dot{F}, ...$
giv	guess of consistent initial values

The following table lists the output arguments for the function symdiff.

linear case	
edif	handle to function `name1`
adif	handle to function `name2`
fdif	handle to function `name3`
nonlinear case	
fdif	handle to function `name1`
dfdif	handle to function `name2`
x0	corrected guess of consistent initial values

Description

`[...] = symdiff(...)` symbolically differentiates functions which are given through their handles. If the user wants to solve a linear DAE he/she has to provide handles to the matrix-valued functions E and A and a handle to the vector-valued function f. Furthermore, he/she has to pass three strings which specify the names of the functions implementing E, A, and f and their derivatives up to the maximal desired order. As output, handles to these functions are returned.

If the user wants to solve a nonlinear problem he/she has to provide a handle to the function specifying F and strings which specify the names of the function implementing F and its derivatives and for the function implementing the Jacobians of $F, \dot{F}, ..., F^{(m)}$ where m is the maximal desired order. As output, handles to these functions are returned. There is a special feature for nonlinear problems. The user can provide a guess for consistent initial values. Using the generated highest derivative of F and the built-in MATLAB function **fsolve** this code tries to correct the guess to get a better starting value for the nonlinear system solver. Note that one has to handle this result with care. It may not be a better starting value than the first guess.

The maximal desired order of differentiation is determined by the parameter MXINDX which the user sets in the GUI and which is stored in the structure array (see section 4).

Note that **symdiff** creates new files in the current directory. In case of linear problems three files are generated. These are named `name1.m`, `name2.m` and `name3.m`. If one is considering nonlinear problems two files are created which are named `name1.m` and `name2.m`. In addition, another file called `hdname1.m` (prefix `'hd'` plus the name chosen by the user) might be created if one used the option to correct guessed initial values. This third file implements the highest desired derivative of F in a way that the function **fsolve** can use it.

In fact all these functions implemented in these files are referenced when a returned handle is evaluated. So deleting or overwriting such a file destroys or changes the according handle, too. Naturally, the automatically generated files to be used with **GELDA** or **GENDA** satisfy the implementation rules imposed by the according Fortran code (see section 6).

- **solvedae**

  solves general DAEs

  Syntax

```
[T, X, xprime, cval, output, civ] = solvedae(s, edif, adif, ...
    fdif, x0)

[T, X, xprime, cval, output, civ] = solvedae(s, edif, adif, ...
    fdif, x0, tspan)

[T, X, xprime, cval, output, civ] = solvedae(s, fdif, dfdif, x0)

[T, X, xprime, cval, output, civ] = solvedae(s, fdif, dfdif, ...
   x0, tspan)
```

  Arguments

  The following table describes the input arguments to the solver solvedae.

s	structure array containing all information which are necessary to setup this solver
linear case	
edif	handle to function specifying E and sufficiently many derivatives of E
adif	handle to function specifying A and sufficiently many derivatives of A
fdif	handle to function specifying f and sufficiently many derivatives of f
nonlinear case	
fdif	handle to a function specifying the left hand side of the inflated DAE
dfdif	handle to a function specifying the Jacobians of the inflated DAE
x0	$n \times 1$ array containing consistent initial values or a guess for these (see section 4.1 and 4.2)
tspan	$1 \times c,\ c > 2$, array containing $t_0,\ t_f$ and further time points at which the solution will be evaluated for performance reasons it is recommended to provide the time points in ascending order

The following table lists the output arguments for the solver solvedae.

T	vector containing the time points where the solution is evaluated if $tspan$ is not given $T = [t_0,\ t_f]$ else $T = tspan$
X	matrix containing the computed solution in which the ith row corresponds to the ith component of x and the jth column corresponds to the jth time point of T
xprime	$n \times 1$ vector which contains the first derivative of x at time t_f
cval	4×1 array containing the characteristic values of the DAE (1) or (2) in the following order $cval(1) =$ strangeness-index $cval(2) =$ number of differential components $cval(3) =$ number of algebraic components $cval(4) =$ number of undetermined components
output	structure array (see section 7)
civ	consistent initial values

Description

`[...] = solvedae(...)` solves differential-algebraic equations of the form

$$E(t)\dot{x}(t) = A(t)x(t) + f(t),\ x(t_0) = x_0$$

or

$$F(t, x(t), \dot{x}(t)) = 0,\ x(t_0) = x_0.$$

The user specifies the kind of DAE (linear or nonlinear) and the way this DAE is solved by calling one of the two setup functions **geldasetup** (for linear problems) or **gendasetup** (for nonlinear problems). Both of these functions return a structure array which contains the necessary general information about the setup. This structure array is the first input to the function **solvedae**. The following inputs are handles to functions which describe the matrix- and vector-valued functions and their derivatives for linear DAEs or the left hand side, its derivatives and its Jacobians for nonlinear DAEs. Initial values which do not necessarily have to be consistent are the next input. As an optional input, time points where the solution should be evaluated can be given. If this optional input is omitted, then the computed solution will be evaluated at t_f only.

2.1 Examples

Example 1

An example for a linear DAE is the following system.

$$\begin{bmatrix} 0 & 0 \\ 1 & -t \end{bmatrix} \dot{x}(t) = -\begin{bmatrix} 1 & -t \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} e^{-t} \\ 0 \end{bmatrix}, \quad t \in [0,1]$$

To solve it with **solvedae** the first step would be to write three m-files implementing the two matrix-valued functions E, A, and the vector-valued function f.

```
function [E] = myE(t)
E(2,2) = -t;
E(2,1) = 1;

function [A] = myA(t)
A(1,2) = t;
A(1,1) = -1;
A(2,1) = 0;
A(2,2) = 0;

function [f] = myf(t)
f(1) = exp(-t);
f(2) = 0;
```

Then one types the following lines in the command window.

```
s = geldasetup(2, 2, [0 1], 'standard');
```

A GUI opens. To use the standard setup of the solver no changes are required. So we accept the setup by pushing the apply-button. As a result we get a structure array called `s` which will be used to call the functions **symdiff** and **solvedae**. Note, that we choose inconsistent initial values. Consistent initial values will be computed by the solver.

```
[edif, adif, fdif] = symdiff(s, @myE, @myA, @myf, ...
    'edif', 'adif', 'fdif');
[T, X] = solvedae(s, edif, adif, fdif, [0;0], [0:0.1:1])
compute consistent initial values and characteristic values...
done
solve DAE...
done

T =

  Columns 1 through 6

         0    0.1000    0.2000    0.3000    0.4000    0.5000

  Columns 7 through 11

    0.6000    0.7000    0.8000    0.9000    1.0000

X =

  Columns 1 through 6

    1.0000    0.9953    0.9825    0.9631    0.9384    0.9098
    1.0000    0.9048    0.8187    0.7408    0.6703    0.6065

  Columns 7 through 11

    0.8781    0.8442    0.8088    0.7725    0.7358
    0.5488    0.4966    0.4493    0.4066    0.3679
```

Using the results `t` and `x` the computed solution can be plotted by typing the following lines.

```
plot(t,x(1,:),'-.k');
hold on;
plot(t,x(2,:),':k');
hold off;
```

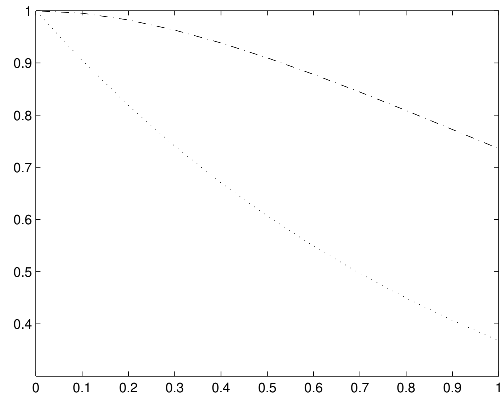

Figure 2: Computed solution of Example 1

Example 2

We want to solve the system

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \dot{x}(t) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 0 \\ e^{-3t} \end{bmatrix}, \quad t \in [0, 1].$$

Because we know that its strangeness-index is 2 and because the derivatives are not difficult to compute we decide to solve it without using symbolic differentiation. Thus, we implement three m-files specifying the functions and their derivatives.

```
function [E, ierr] = myedif(m, n, t, idif)
ierr = 0;
E = zeros(n);
if (idif == 0)
    E(1,1) = 1;
    E(2,2) = 1;
elseif (idif > 0)
    E = zeros(n);
else
    ierr = -2;
end

function [A, ierr] = myadif(m, n, t, idif)
ierr = 0;
A = zeros(n);
if (idif == 0)
    A(1,2) = 1;
```

```
    A(2,3) = 1;
    A(3,1) = 1;
elseif (idif > 0)
    A = zeros(n);
else
    ierr = -2;
end

function [f, ierr] = myfdif(n, t, idif)
ierr = 0;
f = zeros(n,1);
if idif == 0
    f(3) = exp(-3*t);
elseif idif == 1
    f(3) = -3*exp(-3*t);
elseif idif == 2
    f(3) = 9*exp(-3*t);
else
    ierr = -2;
end
```

After calling

```
s = geldasetup(3, 3, [0,1])
```

we use the solver to compute a solution.

```
[T, X, xprime] = solvedae(s, @myedif, @myadif, @myfdif, ...
   [0;0;0], [0:0.1:1])
```

Again, we let **GELDA** compute consistent initial values. We obtain the output below. Using the **plot** command in the shown way yields the Figure 3.

```
T =

  Columns 1 through 6

         0    0.1000    0.2000    0.3000    0.4000    0.5000

  Columns 7 through 11

    0.6000    0.7000    0.8000    0.9000    1.0000
```

```
X =

  Columns 1 through 6

   -1.0000   -0.7408   -0.5488   -0.4066   -0.3012   -0.2231
    3.0000    2.2225    1.6464    1.2197    0.9036    0.6694
   -9.0000   -6.6674   -4.9393   -3.6591   -2.7107   -2.0082

  Columns 7 through 11

   -0.1653   -0.1225   -0.0907   -0.0672   -0.0498
    0.4959    0.3674    0.2722    0.2016    0.1494
   -1.4877   -1.1021   -0.8165   -0.6048   -0.4481


xprime =

    0.1494
   -0.4481
    1.3443
>> plot(T,X(1,:),'-.k');
>> hold on;
>> plot(T,X(2,:),':k');
>> plot(T,X(3,:),'--k');
>> hold off;
```

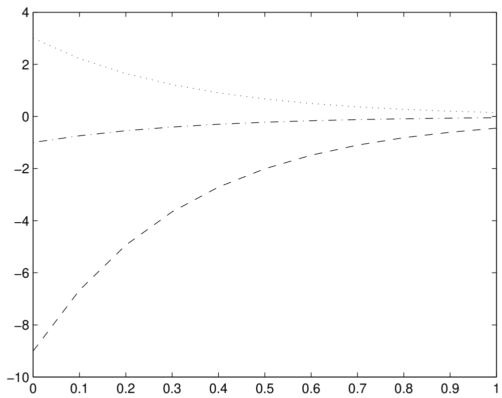

Figure 3: Computed solution of Example 2

Example 3

Consider the nonlinear DAE

$$\begin{bmatrix} \dot{x}_2 - x_1 - e^{t-1} \\ \dot{x}_1(\dot{x}_2 - x_1 - e^{t-1}) + x_2 - t \end{bmatrix} = 0, \quad t \in [0, 1].$$

A valid implementation for this left hand side $F(t, x(t), \dot{x}(t))$ of the DAE would look like this.

```
function [F] = myF(x,t)
expo = exp(t-1);
F(1) = x(4) - x(1) - expo;
F(2) = x(3)*F(1) + x(2) - t;
```

The characteristic values of this problem are $\mu = 1, d_\mu = 0, a_\mu = 2, u_\mu = 0$.

```
s = gendasetup(2, [0 1], [1;0;2;0]);
```

We push the apply-button and call the function **symdiff** for computing symbolically the derivatives and the Jacobians of the left hand side of the DAE. To get a good starting value for consistent initial values we use the feature of **symdiff** that tries to correct given initial values. The approximate starting vector must be of length $(\mathrm{MXINDX} + 2) \cdot n = (1 + 2) \cdot 2 = 6$.

```
>> giv = zeros(6,1);
>> [fdif, dfdif, x0] = symdiff(s, @myF, 'fdif', 'dfdif', giv);
differentiating F...
done
computing Jacobians...
done
correcting initial values...
done
```

Through this we obtain two handles to functions defining the derivatives of F and to its Jacobians. Moreover, we get a vector `x0` that contains a corrected guess for consistent initial values. Note that `x0` does not necessarily contain consistent initial values. They are just another guess. All these will be used when we call the solver **solvedae**.

```
>> [T, X] = solvedae(s, fdif, dfdif, x0, [0:0.1:1])
```

After using the known techniques to plot the computed solution we get a solution that looks like figure 4 shown below.

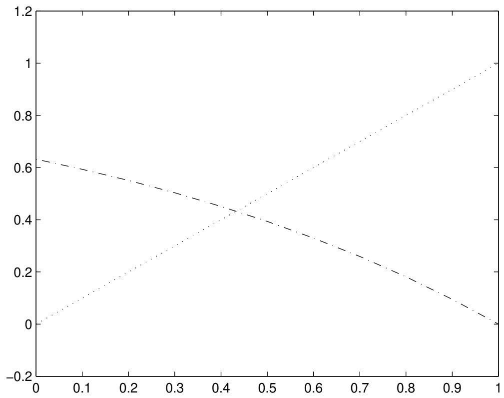

Figure 4: Computed solution of Example 3

3 Setup Structure

This section describes the structure array that is the output of the setup functions **geldasetup** and **gendasetup** and which is an essential input to other functions.

The structure array is the easy-to-use subsumption of certain input arguments that are needed to run the Fortran subroutines. The most important arguments are listed below.

INFO info is an integer array which is used to communicate exactly how the user wants the problem to be solved.

WORKSPACES Two arrays of double precision and integer values are used to provide workspace for the solver and to communicate values for some parameters.

As the user works through the GUI, the `info` and `workspaces` arrays are automatically created and updated according to the actions of the user. Additionally, all entered parameters are saved in fields of the structure array bearing the same name. Here is a list of all fields and their purposes.

name	description
info	info array
m	number of equations
n	number of components of the solution vector x
rw	workspace for double precision numbers
lrw	length of the double precision workspace array
iw	workspace for integer numbers
liw	length of the integer workspace array
t	starting point for the integration
tout	end point of integration
rtol, atol	error tolerances
kind	linear or nonlinear
and all the parameters	
	special fields for **GELDA**
method	integration method (1 = BDF, 2 = Runge-Kutta)
	special fields for **GENDA**
ifix	keeps differential components fixed
scalc	column scaling vector
scalr	row scaling vector
uscal	handle to function for scaling purposes
cval	characteristic values

Table 2: Fields of the Structure Array

Manually changing fields of the structure array not explicitly mentioned in Table 2 does not influence the setup.

4 GUI

Both solvers **GELDA** and **GENDA** offer a couple of parameters that can be changed to adjust the solver to the user's problems. As mentioned in section 2 the setup of a solver is stored in a structure array which is passed to all important functions so that these can handle the problem in the user specified way. The structure arrays for **GELDA** and **GENDA** are different (see section 3).

The two GUIs have some characteristics in common (see figures 5 and 6). One point is the status bar at the bottom of the window. Here information about the user's actions is displayed. For example, if an input value is incorrect an error message will be shown in this line.

Another item is the explanation bar on the right-hand side of the window. To activate any control, the user has to perform a left click on it. After doing so, a short explanation about the purpose of that particular parameter appears in the explanation bar. A second left click marks the according *check box* or enables user input into a *edit text*-control. Once a control has been activated the explanation text can be shown again if the user clicks in a small perimeter around that control. Performing a mouse click directly on that control will not display the explanation again.

Pushing the *apply-pushbutton* causes the GUI to close and all the changes for the setup are saved into the structure array that the user defined as output argument when he called **geldasetup** / **gendasetup**. The current state of the structure array is printed in the command window when the user hits the *print-pushbutton*.

Finally, we turn our focus on the parameters of the GUIs.

4.1 GUI for Linear Problems

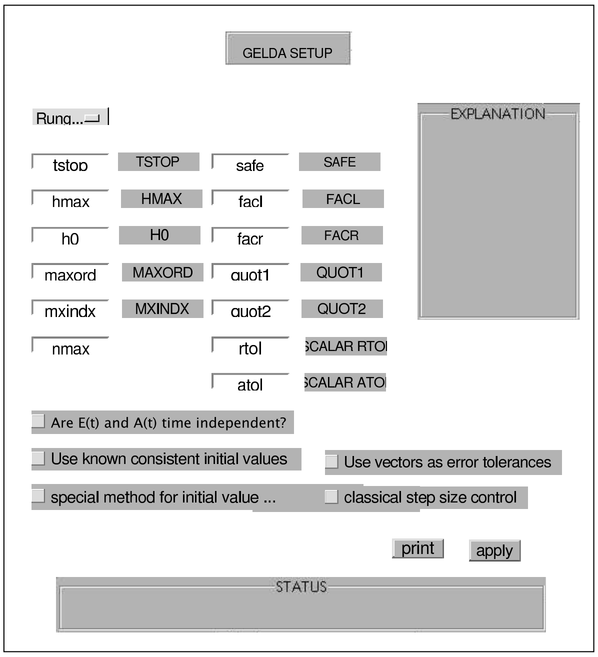

Figure 5: GUI for GELDA

In the top left-hand corner there is a pop-up menu where the method which will be applied to solve the problem can be chosen (BDF- or Runge-Kutta–method).

In the following all parameters which can be set in the GUI are explained. We assume that the structure array holding all information is named `setup`.

TSTOP If the user uses the BDF–solver (METHOD = 1), the code may integrate past TOUT ($= t_f$) and interpolate to obtain the result at TOUT, to handle solutions at many specific values TOUT efficiently. Sometimes it is not possible to integrate beyond some point TSTOP because the equation changes there or it is not defined past TSTOP. Then the user must tell the code not to go past.

The input of the user is stored in `setup.rw(1)`. To indicate that a value for TSTOP is given, `setup.info(4)` is set to 1.

HMAX The user can specify a maximum (absolute value of) step size, so that the code will avoid passing over very large regions.

The input of the user is stored in `setup.rw(2)`. To indicate that a value for HMAX is given `setup.info(7)` is set to 1.

H0 Differential/algebraic problems may suffer from severe scaling difficulties on the first step. If the user knows a great deal about the scaling of his problem, this can be used to alleviate this problem by specifying an initial step size H0.

The input of the user is stored in `setup.rw(3)`. To indicate that a value for H0 is given, `setup.info(8)` is set to 1.

MAXORD If storage is a severe problem and the user uses the BDF–solver (METHOD = 1), memory can be saved by restricting the maximum order MAXORD. The default value is 5. For each order decrease below 5, the code requires fewer locations, however it is likely to be slower. In any case, the user must have $1 \leq$ MAXORD ≤ 5.

The input of the user is stored in `setup.iw(3)`. To indicate that a value for MAXORD is given, `setup.info(9)` is set to 1.

MXINDX The code tries to calculate the strangeness-index of the problem, however it needs more memory for high index problems. The default value for the maximum index MXINDX is 3. The user can decrease it below 3 to save memory (if the strangeness-index of his problem is smaller than 3) or increase it to solve a higher index problem. Note, that EDIF, ADIF and FDIF must provide $E(t)$, $A(t)$, $F(t)$ and (maybe) their first MXINDX derivatives. In any case, the user must have MXINDX ≥ 0.

The input of the user is stored in `setup.iw(4)`. To indicate that a value for MXINDX is given, `setup.info(10)` is set to 1.

NMAX A maximum number of steps NMAX must be specified in order to prevent the code from computing infinitely further in the case of repeated step rejection. The default value for NMAX is 10 000.

The input of the user is stored in `setup.iw(20)`. To indicate that a value for MXINDX is given, `setup.info(14)` is set to 1.

SAFE, FACL, FACR For the Runge–Kutta branch, a safety factor SAFE in step size prediction is used in the formula for calculating the new step size in dependency of the old one and the error norm. The smaller SAFE is chosen, the more the new step size is restricted. SAFE must lie in the interval $[0.001, 1]$. The default value is SAFE $= 0.9$. Furthermore, parameters FACL, FACR for step size selection restrict the relation between the old and the new step size. The new step size is chosen subject to $1/\text{FACL} \leq \text{HNEW}/\text{HOLD} \leq 1/\text{FACR}$. The default values are FACL $= 5.0$ and FACR $= 0.125$.

The input of the user is stored in the following fashion. The factor SAFE is saved in `setup.rw(11)`, FACL in `setup.rw(12)` and FACR is saved in `setup.rw(13)`. To indicate that some factor has been input, `setup.info(16)` is set to 1.

QUOT1, QUOT2 For the Runge–Kutta branch, if HNEW is not far from HOLD (QUOT1 $<$ HNEW/HOLD $<$ QUOT2) and the matrices E and A are constant, work can be saved by setting HNEW = HOLD and using the system matrix of the previous step, so that a new LU–decomposition is not necessary. For small systems one may have QUOT1 $= 1.0$, QUOT2 $= 1.2$, for large full systems QUOT1 $= 0.99$, QUOT2 $= 2.0$ might be good. Default values are QUOT1 $= 1.0$, QUOT2 $= 1.2$.

The input of the user is stored in the following fashion. The factor QUOT1 is saved in `setup.rw(14)` and QUOT2 is saved in `setup.rw(15)`. To indicate that some factor has been given, `setup.info(17)` is set to 1.

RTOL, ATOL The relative and absolute error tolerances which the user provides to indicate how accurately he wishes the solution to be computed. The user may choose RTOL and ATOL to be both scalars or else both vectors.

Scalar error tolerances (`setup.inf(2) = 0`) are entered via the *edit text* controls of the GUI. Vector error tolerances (`setup.info(2) = 1`) are entered via the function **supply** by typing the following commands at the MATLAB prompt.

```
rtol = [...];
atol = [...];
setup = supply('rtol', rtol, 'atol', atol);
```

The intention to supply error tolerances as vectors is announced by marking the appropriate *check box* of the GUI.

The tolerances are used by the code in a local error test at each step which requires roughly that

$$|\text{LOCAL ERROR}| \leq \text{RTOL} * |\text{X}| + \text{ATOL}$$

for each vector component.

The true (global) error is the difference between the true solution of the initial value problem and the computed approximation. Practically all present day codes, including this one, control the local error at each step and do not even attempt to control the global error directly.

Usually, but not always, the true accuracy of the computed X is comparable to the error tolerances. This code will usually, but not always, deliver a more accurate solution if the user reduces the tolerances and integrate again. By comparing two such solutions, the user can get a fairly reliable idea of the true error in the solution at the bigger tolerances.

Setting ATOL=0 results in a pure relative error test on that component. Setting RTOL=0 results in a pure absolute error test on that component. A mixed test with non-zero RTOL and ATOL corresponds roughly to a relative error test when the solution component is much bigger than ATOL and to an absolute error test when the solution component is smaller than the threshold ATOL.

The code will not attempt to compute a solution at an accuracy unreasonable for the machine being used. It will advise the user if the required accuracy is too much and inform the user as to the maximum accuracy it believes to be possible.

Underneath the *edit text*-controls some *check boxes* allow further customization of the setup.

time independent The code assumes that the user wants to solve a time dependent problem. If $E(t)$ and $A(t)$ are both constant the user can speed up the code.

To indicate this, `setup.info(5)` is set to 1.

known initial values In this code it is not necessary to provide consistent initial conditions. Using the special structure of the strangeness free DAE, the code can compute consistent initial values to start the integration. However, often consistent initial values are known and the code should use these values.

To indicate this, `setup.info(11)`is set to 1.

special method If INFO(11) = 0, the code computes consistent initial values in the least squares sense. The default method is to compute consistent initial values which are close (in the least squares sense) to the given X_0. Sometimes the user knows which are the differential variables and wants to prescribe these variables. In this case, the user can use a different method, which keeps the differential variables fixed.

To indicate this, `setup.info(12)` is set to 1.

classical step size control For the Runge–Kutta branch, the user can choose between two step size strategies. If not specified otherwise, the code will use the modified predictive controller of Gustafsson, which seems to produce safer results. As alternative for simple problems, the user can apply the classical step size control which produces often slightly faster runs.

To indicate this, `setup.info(15)` is set to 1.

4.2 GUI for Nonlinear Problems

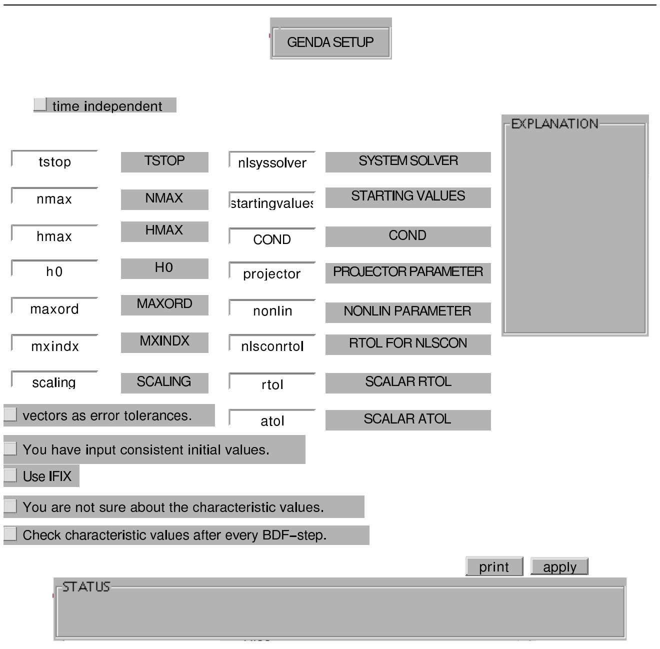

Figure 6: GUI for GENDA

In the top left-hand corner there is a *check box* which should be marked if the function F is not dependent directly on t.

Underneath, there are several *edit text*-controls allowing to change the parameters of **GENDA**. The purpose of each parameter is explained below. Again we assume that the structure array storing the setup is called `setup`.

TSTOP The code may integrate past TOUT and interpolate to obtain the result at TOUT ($= t_f$), to handle solutions at many specific values TOUT efficiently. Sometimes it is not possible to integrate beyond some point

TSTOP because the equation changes there or it is not defined past TSTOP. Then the user must tell the code not to go past.

The input of the user is stored in `setup.rw(1)`. To indicate that a value for TSTOP is given, `setup.info(4)` is set to 1.

NMAX A maximum number of steps NMAX must be specified in order to prevent the code from computing infinitely further in the case of repeated step rejection. The default value for NMAX is 10 000.

The input of the user is stored in `setup.iw(20)`. To indicate that a value for NMAX is given, `setup.info(6)` is set to 1.

HMAX The user can specify a maximum (absolute value of) step size, so that the code will avoid passing over very large regions.

The input of the user is stored in `setup.rw(2)`. To indicate that a value for HMAX is given, `setup.info(7)` is set to 1.

H0 Nonlinear DAEs sometimes suffer from severe scaling difficulties on the first step. If the user knows a great deal about the scaling of his problem, this can be used to alleviate this problem by specifying an initial step size H0.

The input of the user is stored in `setup.rw(3)`. To indicate that a value for H0 is given, `setup.info(8)` is set to 1.

MAXORD If storage is a severe problem, then one can save some memory by restricting the maximum order MAXORD of the BDF-method. The default value is 5. For each order decrease below 5, the code requires fewer locations, however it is likely to be slower. In any case, one must have 1 $\leq$ MAXORD $\leq$ 5.

The input of the user is stored in `setup.iw(3)`. To indicate that a value for MAXORD is given, `setup.info(9)` is set to 1.

MXINDX If the user wishes, the code tries to check the strangeness-index μ and the other characteristic values of the problem. The default value for the maximum index MXINDX is μ=CVAL(1). The user can increase it if the index of the problem may be higher. Note, that one must provide $F(t, x(t), \dot{x}(t))$, the first MXINDX derivatives of F and the partial derivatives with respect to $\dot{x}, ..., x^{(\mathrm{MXINDX}+1)}$ if one is not using the function **symdiff**. In any case, the user must have MXINDX ≥ 0.

The input of the user is stored in `setup.iw(4)`. To indicate that a value for MXINDX is given, `setup.info(10)` is set to 1.

SCALING Nonlinear DAEs sometimes suffer from severe scaling problems. DGENDA uses row- and column scaling of the iteration matrices, where the scaling vectors SCALR and SCALC are automatically updated. If the problem does not suffer from scaling problems, then the user can disable this automatic scaling. If, on the other hand, the user knows a lot about the correct scaling of the problem, it is possible to supply a user-defined scaling routine USCAL. One has to use the function **supply** to store the function handle to USCAL and the scaling vectors in the structure array. For example, one writes

```
uh = @myUsal;
scalc = [...];
scalr = [...];
setup = supply('uscal', uh, 'scalc', scalc, ...
    'scalr', scalr);
```

to supply these data.

To indicate that automatic scaling is not used, `setup.info(13)` is set to 1. The value of `setup.iw(16)` is set to 0 if one wants to disable any scaling or `setup.iw(16)` is set to 2 if one wants to use ones own scaling function.

NLSYSSOLVER In every step the BDF-method uses the simplified Gauss-Newton method to solve the nonlinear system. If μ=0, then one can let the code decide when a new iteration-matrix is needed. The method used for this decision is the same as in DASSL [11]. One can also let the code use the classical Gauss-Newton method, which is much slower than the default method.

To indicate that the simplified Gauss-Newton method is not used, `setup.info(14)` is set to 1. The value of `setup.iw(2)` is set to 0 if one prefers the method used in DASSL or `setup.iw(2)` is set to 2 if one wants the code to use the classical Gauss-Newton method in every step.

STARTINGVALUES In every step of the BDF-solver the code obtains the starting values for X(1 : N) and XPRIME for the Gauss-Newton iterations by interpolation. For the remaining starting values X(N+1 : (μ +2)N) it takes by default X(N+1 : 2N)=XPRIME and classical homotopy for the remaining values, i.e. it takes the last solutions as starting values for the next step.

To indicate this changed method for obtaining starting values, `setup.info(15)` is set to 1. The value of `setup.iw(17)` is set to 0 if one wants to use classical homotopy also for X(N+1:2N), `setup.iw(17)` is set to 2 if one wants the code to compute the starting values for X(N+1:2N) by linear extrapolation, or `setup.info(17)` is set to 3 if one wants the code to obtain all values X(N+1:(μ +2)N) by linear extrapolation.

COND If the user does not know the characteristic values of the problem, the rank decisions made to compute the characteristic values may suffer from scaling problems and of the error made in every BDF-step. A singular value of a matrix is taken to be zero, if it is smaller than the largest singular value multiplied by $COND^{-1}$. By default, we set $COND = 10^{12}$.

The input of the user is stored in `setup.rw(1)`. To indicate that the value of COND is given, `setup.info(18)` is set to 1.

PROJECTOR The code assumes that the user wants to solve a general nonlinear problem. If it is known that the projector Z_2 for the algebraic equations does only depend on t, then the code can compute a reduced system. This will result in a faster run of the program. If the problem is linear the user can also help the code to be faster.

To indicate this `setup.info(20)` is set to 1. If Z_2 only depends on t, `setup.iw(23)` is set to 1 or if the problem is linear, `setup.iw(23)` is set to 0.

NONLIN The code uses the subroutine NLSCON [9] for the calculation of consistent initial values if this computation is necessary. By default the NONLIN parameter of NLSCON is set to 2 if the DAE is nonlinear and to 1 if the problem is linear. The user can change this value if DGENDA fails to compute consistent initial values.

To indicate that the NONLIN-parameter is changed, `setup.info(21)` is set to 1. The value of the NONLIN-parameter is stored in `setup.iw(24)`.

NLSCONRTOL If consistent initial values must be computed, the user can prescribe the required relative precision of the consistent initial values by setting the RTOL parameter of the subroutine NLSCON. By default, RTOL is set to 10^{-10}.

To indicate that NLSCONRTOL has changed, `setup.info(22)` is set to 1. The value of NLSCONRTOL is stored in `setup.rw(11)`.

RTOL, ATOL The relative and absolute error tolerances which the user provides to indicate how accurately the solution should be computed. The user may choose RTOL and ATOL to be both scalars or else both vectors.

Scalar error tolerances (`setup.inf(2) = 0`) are entered via the *edit text* controls of the GUI. Vector error tolerances (`setup.info(2) = 1`) are entered via the function **supply** by typing the following commands at the MATLAB prompt.

```
rtol = [...];
atol = [...];
setup = supply('rtol', rtol, 'atol', atol);
```

One announces the intention to supply error tolerances as vectors by marking the appropriate *check box* of the GUI.

The tolerances are used by the code in a local error test at each step which requires roughly that

$$|\text{LOCAL ERROR}| \leq \text{RTOL} * |\text{X}| + \text{ATOL}$$

for each vector component.

The true (global) error is the difference between the true solution of the initial value problem and the computed approximation. Practically all present day codes, including this one, control the local error at each step and do not even attempt to control the global error directly.

Usually, but not always, the true accuracy of the computed X is comparable to the error tolerances. This code will usually, but not always, deliver a more accurate solution if the user reduces the tolerances and integrate again. By comparing two such solutions the user can get a fairly reliable idea of the true error in the solution at the bigger tolerances.

Setting ATOL=0 results in a pure relative error test on that component. Setting RTOL=0 results in a pure absolute error test on that component. A mixed test with non-zero RTOL and ATOL corresponds roughly to a relative error test when the solution component is much bigger than ATOL and to an absolute error test when the solution component is smaller than the threshold ATOL.

The code will not attempt to compute a solution at an accuracy unreasonable for the machine being used. It will advise the user if the required accuracy is too hard and inform the user as to the maximum accuracy it believes to be possible.

Now we proceed with the *check boxes* and explain their purposes.

consistent initial values In this code it is not necessary to provide consistent initial conditions. Using the special structure of the strangeness free DAE, the code can compute consistent initial values to start the integration. However, often consistent initial values are known and the code should use these values.

If consistent initial values are known, mark this *check box*. The value of `setup.info(11)` is set to 1.

IFIX If `setup.info(11)` = 0, the code computes consistent initial values in the least squares sense. The default method is to compute consistent initial values which are close (in the least squares sense) to the given X. Sometimes the user knows which are the differential variables and wants to prescribe these variables. In this case, the user can use a different method, which keeps some of the variables fixed. Note, that this may lead to a rank-deficient Jacobian.

One should mark this *check box* if one wants to use IFIX and provide the corresponding vector via the function supply by typing

```
ifix = [...];
setup = supply('ifix', ifix);
```

at the prompt in the MATLAB command window. The value of `setup.info(12)` is set to 1.

correct characteristic values The user must initialize the code with correct characteristic values of the DAE. However, DGENDA can try to calculate these values after a successful BDF-step or after consistent initial values have been computed. If it detects any changes in the characteristic values the code will return with an error flag. This is not necessary if the user is sure about the given characteristic values.

To indicate that one is not sure about the characteristic values that one inputs when calling the function **gendasetup** mark this *check box*. Doing so changes the value of `setup.info(16)` to 1.

characteristic value check If `setup.info(16)` = 1, one can let the code check the characteristic values once after every call of DGENDA or after every BDF-step.

If this *check box* is marked the characteristic values will be checked after every BDF-step (`setup.info(17)` = 1). They will be checked after every call of DGENDA if one leaves the *check box* unmarked (`setup.info(17)` = 0).

5 Symbolic Differentiation

This toolbox features the function **symdiff** that is capable of symbolic differentiation. To achieve this it uses the symbolic math toolbox of MATLAB. Fortunately, the user does not need to know something about this particular toolbox. It is sufficient to provide a standard m-file that implements the considered function. Among other inputs, a handle to this function is passed to **symdiff**. Inside this function the standard function is transformed into a symbolic function for which the mentioned toolbox calculates the symbolic derivatives. Then these are transformed into strings which are written into a new standard m-file such that they can be used easily. See section 6 for more information about the requirements of these functions. Of course, these requirements are fulfilled by the automatically generated file.

However, this comfort is computationally expensive. When writing a function for $F(t, x(t), \dot{x}(t))$ that will be passed to **symdiff** one has to stick to two conventions.

1. The components of the input vector for such a function must be ordered in a certain way.

2. The first assignment of a component of F must contain a variable if any exist, that means if F is not constant.

The function must take two arguments exactly. The first one specifies x and $\dot{x}$ and the second one represents t. Let x be the solution vector of (2). Let n be its dimension. Then the first input argument is a vector $\tilde{x}$ of length $2n$ where the first n elements correspond to x itself and the elements $\tilde{x}(n+1), \tilde{x}(n+2), ..., \tilde{x}(2n)$ corresponds to $\dot{x}(1), \dot{x}(2), ...\dot{x}(n)$.

Assume that F is not constant, i.e., some components depend on t or $\tilde{x}$. The first assignment to that vector that is returned by the user supplied function must contain at least one of the variables $t, \tilde{x}(1), \tilde{x}(2), ..., \tilde{x}(2n)$. For Example, let F be two dimensional and defined like this.

$$F := \begin{bmatrix} 0 \\ \dot{x}_2 x_1 + \dot{x}_1 x_2 + t \end{bmatrix}$$

Then the user supplied M-file should look like that.

```
function [F] = myF(x,t)
F(2) = x(4)*x(1) + x(3)*x(2) + t;
F(1) = 0;
```

If the user started with `F(1) = 0` instead, then MATLAB would assume that the user wants to create a double precision matrix. So MATLAB will attempt to convert the symbolic variables at the assignment of `F(2)` to double precision numbers. That is not possible because there is no value associated with symbolic variables. Trying to convert a symbolic variable to a number causes MATLAB to display the following exception.

```
??? Conversion to double from sym is not possible.
```

The same problem occurs in the linear case. In the implementations of the functions $E(t)$, $A(t)$ and $f(t)$ the first assignment to the matrix which is returned must depend on t. So a valid implementation of the matrix-valued function

$$A(t) = \begin{bmatrix} -1 & t \\ 0 & 0 \end{bmatrix}$$

would look like this, for example.

```
function [A] = myA(t)
A(1,2) = t;
A(1,1) = -1;
A(1,2) = 0;
A(2,2) = 0;
```

The second convention implies that a user must not use the built-in MATLAB functions `zeros` or `ones` because initializing a matrix with these functions turns it into a double precision matrix which cannot be transformed into a symbolic matrix. It is very important to keep that in mind when one implements a function. Convention 2 can be ignored when dealing with a time independent problem.

Consider the following example. The constant matrix-valued function

$$E(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

can be implemented like this.

```
function [E] = myE(t)
E = zeros(3);
E(1,1) = 1;
E(2,2) = 1;
```

6 User Supplied Functions

If the derivatives are known or can be computed easily, then it is often too time consuming to use **symdiff**. Instead, the user can supply functions implementing the given mapping and its derivatives or its Jacobians in the nonlinear case. The number and order of the input arguments is imposed by the Fortran codes.

This section also covers the requirements that a function must meet for scaling purpose.

Matrix-valued functions for linear problems must be of the following form.

- **Header**

  `function [G,ierr] = anyName(l, n, t, idif)`

  Arguments

  The following table describes the input arguments to the auxiliary function.

l	number of equations
n	number of variables
t	time
idif	parameter specifying the order of the desired derivative.

  The following table lists the output arguments for the auxiliary function.

G	`idif`-th derivative of G
ierr	error flag being always zeros only if `idif` is larger than the highest derivative that this function provides *ierr* is assigned the value -2.

  Description

  This function takes as input the number of equations m, the number of variables n, the time t and the parameter `idif`. As output, it produces the `idif`-th derivative of the function at time t. An error is signaled via *ierr* if a higher derivative is requested than provided. Note that one has to provide such functions for E and A from (1).

The vector-valued functions of linear problems require implementation according to the following standards.

- **Header**

  `function [f,ierr] = anyName(l, t, idif)`

  Arguments

  The following table describes the input arguments to the auxiliary function.

l	number of equations
t	time
idif	parameter specifying the order of desired derivative.

  The following table lists the output arguments for the auxiliary function.

f	`idif`-th derivative of G
ierr	error flag being always zeros only if `idif` is larger than the highest derivative that this function provides *ierr* is assigned the value -2.

  Description

  This function takes as input the number of equations m as well as the time t and the parameter `idif`. As output, it produces the `idif`-th derivative of f at time t. An error is signaled via *ierr* if a higher derivative is requested than provided.

The functions for nonlinear problems look a little bit different. It is shown below. In the following let μ be the strangeness-index of the DAE.

- **Header**

  `function [F,ierr] = anyName(t, idif, x)`

  Arguments

  The following table describes the input arguments to the auxiliary function.

t	time
idif	parameter specifying the order of desired derivative
x	vector containing an approximation to the solution $\tilde{x} = (x, \dot{x}, ..., x^{(m+1)})$, where $m \geq \mu$.

  The following table lists the output arguments for the auxiliary function.

F	column vector containing the `idif`-th derivative of F
ierr	error flag being always zeros only if `idif` is larger than the highest derivative that this function provides *ierr* is assigned the value -1.

Description

This function takes as input the time t, the parameter $idif$ and the vector x containing $(x, \dot{x}, ..., x^{(m+1)})$, where $m \geq \mu$. As output, the function produces the `idif`-th derivative of the DAE at time t. An error is signaled via $ierr$ if a higher derivative is requested than provided.

Additionally, a function providing the Jacobians of the inflated DAE is needed. Its requirements are shown below.

- **Header**

  `function [J, ierr] = anyName2(t, idif, x)`

  ### Arguments

  The following table describes the input arguments to the auxiliary function.

t	time
idif	parameter specifying the order of desired derivative
x	vector containing an approximation to the solution $\tilde{x} = (x, \dot{x}, ..., x^{(m+1)})$, where $m \geq \mu$.

  The following table lists the output arguments for the auxiliary function.

J	Jacobian of the `idif`-th derivative of F
ierr	error flag being always zeros only if `idif` is larger than the highest derivative that this function provides $ierr$ is assigned the value -1.

  ### Description

  The function takes as input the time t, the parameter `idif` and the vector x which contains an approximation to the solution $\tilde{x} = (x, \dot{x}, ..., x^{(m+1)})$, where $m \geq \mu$. As output, it produces all partial derivatives of the `idif`-th derivative of $F(t, x(t), \dot{x}(t))$ with respect to all elements of x.

A function that realizes user defined scaling must be of the following form.

- **Header**

  `function [B, ierr] = uscal(A, scalc, scalr)`

  ### Arguments

  The following table describes the input arguments to the scaling function.

A	matrix to be scaled
scalc	column scaling vector
scalr	row scaling vector.

  The following table lists the output arguments for the scaling function.

B	scaled matrix `B = diag(scalr)*A*diag(scalc)`
ierr	error flag being always zero only if the scaling vectors have incorrect sizes *ierr* is assigned the value -1.

Description

This function takes as input the matrix $A \in \mathbb{R}^{mq \times nq}$ and the vectors $scalc \in \mathbb{R}^{nq}$ and $scalr \in \mathbb{R}^{mq}$. As output, the function produces column scale factors *scalc* and column scale factors *scalr* and a matrix B such that $B = \mathrm{diag}(scalr) A \mathrm{diag}(scalc)$. The integer error flag IERR should be set to a negative value if there was any illegal input.

7 Output Structure

The output structure array contains the following information. Please note that the field `contf` exists only in the nonlinear case.

orderns	order of the method to be attempted on the next step
orderls	oder of the method used on the last step
steps	steps taken so far
feval	number of calls of function evaluations
fac	number of the factorizations of the system matrix so far
errtf	total number of error test failures
contf	total number of convergence test failures
h	step size to be attempted on the next step
tend	farthest time point integration has reached
hend	step size used on the last successful step

8 Handling Large Problems

The Fortran codes use arrays as workspaces. Here all the parameters are stored and calculations are carried out. Two workspace arrays are needed in **GELDA** and **GENDA**. One is used for integer numbers and one for double precision numbers. Of course, the needed length of a workspace array depends on the considered problem.

During the initialization of the variables and parameters in the MEX-file the exact dimensions of these different items which are passed from MATLAB are not known. Although it is possible to circumvent this difficulty by dynamically allocating memory in a Fortran MEX-file, the toolbox does not use this technique. Rather, the sizes of the problematic items are declared using a fixed upper bound (50,000) which is set as a parameter. So one does not have to bother about this topic. But a problem remains. Large problems require large workspaces. Their lengths might exceed the upper bound. In that case the toolbox will signal a warning. To try to solve such a problem anyway one can increase the upper bound by changing the value of the parameter in the Fortran code of the MEX-file indicated by the warning. After this has been done one has to compile the code again. Run the M-script `compile` in the directory where the MEX-files **gelda.f** and **genda.f** and the folder **FSP** are saved.

Attention

Use the command **mex** (refer to the MATLAB documentation) to properly compile the toolbox. Especially, use the option `mex -setup` to pick the right compiler and consult the online reference

```
http://www.mathworks.com/access/helpdesk/help/techdoc/...
matlab_external/matlab_external.html
```

to learn how to adjust the script `compile` to the system in use. If one is not connected to the Internet the same information can be found in the MATLAB Help under the keyword

```
LAPACK and BLAS functions::building MEX files for
```

Nevertheless, one should keep in mind that MATLAB is probably not an appropriate tool to solve problems of such a huge dimension. So, before recompiling the toolbox one should think about addressing this problem in another way.

9 Mathematical Background

The main part of the solvers **lindaesolve** and **nldaesolve** are the two Fortran subroutines **dgelda** and **dgenda**. This section gives a very short introduction on how they work.

DGELDA [7] solves linear differential algebraic equations (DAEs) with variable coefficients of the form

$$\begin{aligned} E(t)\dot{x}(t) &= A(t)x(t) + f(t) \\ x(t_0) &= x_0 \end{aligned}$$

for x in a specified range of the independent variable t.

The most important invariant in the analysis of linear DAEs is the so called *strangeness-index*, which generalizes the differentiation-index [2] for systems with undetermined components.

The implementation of DGELDA [7] is based on the construction of the discretization scheme introduced in [5], which first determines all the local invariants and then transforms the system into a strangeness-free DAE with the same solution set.

The strangeness-free DAE is solved by either BDF methods, which were adapted from DASSL of Petzold [11], or a Runge–Kutta method, which was adapted from RADAU5 of Hairer/Wanner [4].

DGENDA [8] solves general differential algebraic equations (DAEs) of the form

$$\begin{aligned} F(t, x(t), \dot{x}(t)) &= 0, \\ x(t_0) &= x_0, \end{aligned} \tag{3}$$

for x in a specified range $[t_0, t_f]$ of the independent variable t. No restrictions on the strangeness-index are needed.

We need information about several derivatives of the given DAE. For this we denote by

$$F_l(t, x, \dot{x}, ..., x^{(l+1)}) = \begin{bmatrix} F(t, x, \dot{x}) \\ \frac{dF}{dt}(t, x, \dot{x}, \ddot{x}) \\ \vdots \\ \frac{d^l F}{dt^l}(t, x, \dot{x}, ..., x^{(l+1)}) \end{bmatrix} = 0 \tag{4}$$

the *inflated* nonlinear DAE obtained by successive differentiation and we denote by

$$\begin{aligned} M_l(t, x, \dot{x}, ..., x^{(l+1)}) &= F_{l;\dot{x},...,x^{(l+1)}}(t, x, \dot{x}, ..., x^{(l+1)}), \\ N_l(t, x, \dot{x}, ..., x^{(l+1)}) &= -(F_{l;x}(t, x, \dot{x}, ..., x^{(l+1)}), 0, ..., 0) \end{aligned}$$

its Jacobians. The DAE has to satisfy the following Hypothesis.

Hypothesis 1 *There exist integers $\hat{\mu}$, $\hat{d}$ and $\hat{a}$, such that for all values $(t, x, \dot{x}, ..., x^{(\hat{\mu}+1)}) \in \mathbb{L}$ with*

$$\mathbb{L} = \{(t, x, \dot{x}, ..., x^{(\hat{\mu}+1)}) \in \mathbb{I} \times \mathbb{R}^n \times \mathbb{R}^n \times ... \times \mathbb{R}^n | F_{\hat{\mu}}(t, x, \dot{x}, ..., x^{(\hat{\mu}+1)}) = 0\}$$

associated with F the following properties hold

1. *We have* $\operatorname{rank} M_{\hat{\mu}}(t, x, \dot{x}, ..., x^{(\hat{\mu}+1)}) = (\hat{\mu} + 1)n - \hat{a}$, *and there exists a matrix function $\hat{Z}_2$ being smooth on $\mathbb{L}$ with orthonormal columns and size $((\hat{\mu} + 1)n, \hat{a})$ satisfying $\hat{Z}_2^T M_{\hat{\mu}} = 0$.*

2. *We have* $\operatorname{rank} \hat{A}_2(t, x, \dot{x}, ..., x^{(\hat{\mu}+1)}) = \hat{a}$, *where $\hat{A}_2 = \hat{Z}_2^T N_{\hat{\mu}}[I_n 0 \cdots 0]^T$, and there exists a matrix function $\hat{T}_2$ being smooth on $\mathbb{L}$ with orthonormal columns and size $(n, \hat{d})$, $\hat{d} = n - \hat{a}$, satisfying $\hat{A}_2 \hat{T}_2 = 0$.*

3. *We have* $\operatorname{rank} F_{\dot{x}}(t, x, \dot{x}) \hat{T}_2(t, x, \dot{x}, ..., x^{(\hat{\mu}+1)}) = \hat{d}$, *and there exists a matrix function $\hat{Z}_1$ being smooth on $\mathbb{L}$ with orthonormal columns and size $(n, \hat{d})$ yielding that $\hat{E}_1 = \hat{Z}_1^T F_{\dot{x}}(t, x, \dot{x})$ has constant rank $\hat{d}$.*

When the DAE (3) satisfies Hypothesis 1, then the *(global) strangeness-index μ* of (3) is defined as the least possible $\hat{\mu}$ for which the above properties hold. The corresponding numbers d and a are the numbers of differential and algebraic equations of the DAE.

If the *differentiation-index* (see, e.g., [2]) is well-defined, Hypothesis 1 is always satisfied, and the strangeness index is zero if the differentiation-index is zero and equal to the differentiation-index lowered by 1 otherwise.

In [6] it has been shown that every sufficiently smooth solution x^* of (3), where F satisfies Hypothesis 1, is a locally unique solution of a problem of the form

$$\hat{Z}_1^T F(t, x_1, x_2, \dot{x}_1, \dot{x}_2) = 0, \tag{5}$$

$$x_2 = \hat{G}_2(t, x_1), \tag{6}$$

which has a vanishing strangeness-index, i.e. differentiation-index at most one.

Consistency of an initial value means that (6) is satisfied while arbitrary initial values can be chosen for the differential variables x_1. In [6] it has been shown that every $y_0 \in \mathbb{L}$ can be locally extended to a solution of (3). Thus every part (t_0, x_0) of $y_0 \in \mathbb{L}$ is a consistent initial value. Therefore, to determine a consistent initial value one must solve

$$F_\mu(t_0, x, \dot{x}, ..., x^{(\mu+1)}) = 0 \tag{7}$$

for $(x, \dot{x}, ..., x^{(\mu+1)})$. The solution of this underdetermined system of nonlinear equations is computed in a least squares sense with the subroutine NLSCON (see [9]) which is an implementation of the Gauss-Newton method [3]. The user must supply a guess of the initial values as a starting value for the Gauss-Newton iteration.

The initial values for the d differential variables can be set to any value, similar to the case of an ODE. The user can choose these variables to be kept fixed during the Gauss-Newton iterations by setting up the IFIX-array. In this case, the corresponding columns of the Jacobian are set to zero. Note that this will lead to a rank drop of the Jacobian if any of the algebraic variables are fixed. In this case the code will return an error message.

DGENDA uses a BDF method with order and step size control. The BDF solver used here is an adaption of the code implemented in DASSL (see [11]) for solving DAEs of index at most one. Before discretizing the DAE at a time step proceeding from t_0 to $t_0 + h$ we have to compute a locally equivalent system similar to (5),(6). In every step we then solve a system of nonlinear equations of the form

$$\begin{aligned} F_\mu(t_0 + h, x, \dot{x}, ..., x^{(\mu+1)}) &= 0, & (8)\\ \tilde{Z}_1^T F(t_0 + h, x, \alpha x + \beta) &= 0, & (9) \end{aligned}$$

where $\tilde{Z}_1$ denotes some approximation to $\hat{Z}_1$ at the desired solution. Equation (8) yields a solution for which (6) holds, so that the algebraic constraints are satisfied and (9) is a discretization of (5).

The solution is computed with a Gauss-Newton method, see e.g. [3]. An initial guess $\bar{x}_0 = (x_0, \dot{x}_0, ..., x_0^{(\mu+1)})$ is iteratively improved by a correction $\Delta\bar{x}_i$, i.e.

$$\bar{x}_{i+1} = \bar{x}_i + \Delta\bar{x}_i, \qquad i = 0, 1, ... \quad , \tag{10}$$

where

$$\Delta\bar{x}_i = -\bar{J}_i^+ \bar{F}(t_0 + h, \bar{x}_i). \tag{11}$$

Here, $\bar{F}$ denotes the system given by (8), (9),

$$\bar{J}_i = \begin{bmatrix} -N_\mu[I_n 0 \cdots 0] & M_\mu \\ \tilde{Z}_1^T(F_x + \alpha F_{\dot{x}}) & 0 \end{bmatrix} \tag{12}$$

is its Jacobian at state $\bar{x}_i$ and $\bar{J}_i^+$ the Moore-Penrose pseudo inverse of $\bar{J}_i$ (see e.g. [1]). Note that the Jacobian of (8), (9) is generally nonsquare but has full row rank at every solution $(x, \dot{x}, ..., x^{(\mu+1)})$ of (3) if $\tilde{Z}_1$ is a sufficiently good approximation to $\hat{Z}_1$ and the step size is sufficiently small. This property extends to a neighborhood of the solution set, thus we get quadratic convergence to a solution and we can apply a simplified Gauss-Newton method [10] by fixing the Jacobian at any time step.

By default, DGENDA uses the simplified Gauss-Newton method [10] with an initial $\bar{x}_0$ where x_0 and $\dot{x}_0$ are obtained by evaluating a predictor polynomial at time t_0+h and $(\ddot{x}_0, ..., x_0^{(\mu+1)})$ is obtained by classical homotopy [10], i.e. the results at the previous time step t_0 are chosen as initial values at $t_0 + h$. Optionally, the user can force the code to use the classical Gauss-Newton method, where the Jacobian is reevaluated at every iteration or to use the approach implemented in DASSL, where the Jacobian is reevaluated after several time steps when a certain criterion is fulfilled. Furthermore, the user can tell the code to compute some of the components of the initial solution vector $\bar{x}_0$ by linear extrapolation.

The corrector iterations (10) are terminated if an estimate for the local relative and absolute error is small enough. This error test requires roughly that

$$|\text{LOCAL ERROR}| \leq \text{RTOL} * |\text{X}| + \text{ATOL} \tag{13}$$

for each component of the solution vector $(x, \dot{x}, ..., x^{(\mu+1)})$ at every time step. RTOL and ATOL can be scalars, such that (13) must hold for every component, or they can be vectors of size $(\mu + 2)n$, such that the user can set different tolerances for every component of the solution.

If it is known that $\hat{Z}_2$ of Hypothesis 1 only depends on t, then one can choose $\tilde{Z}_2 = \hat{Z}_2$ and equation (8) can be reduced to

$$\tilde{Z}_2^T F_\mu(t_0 + h, x, \dot{x}, ..., x^{(\mu+1)}) \quad = \quad 0. \tag{14}$$

Due to Hypothesis 1 the system (14), (9) does only depend on x and is uniquely solvable. This approach is also implemented in the code but no check will be made whether $\hat{Z}_2$ is independent of $(x, \dot{x}, ..., x^{(\mu+1)})$.

By default, the user has to set the characteristic values μ, d, a and u to their correct values ($u = n - a - d$ must be zero in this version of DGENDA). Note that the rank assumptions in Hypothesis 1 only hold for $y_0 \in \mathbb{L}$, such that they can be violated at $(t_0 + h, \bar{x}_0)$, where $\bar{x}_0$ is the initial solution vector for the Gauss-Newton iterations. Thus, in general, it is not possible to detect these values when computing the projectors $\tilde{Z}_1$, $\tilde{T}_2$ and $\tilde{Z}_2$. It can also be difficult to apply the approach used for the linear solver DGELDA [7] to a linearization of (3) after a successful iteration of the BDF-solver because the computed approximation to the solution generally lies in a neighborhood of $\mathbb{L}$ prescribed by the tolerances, but not in $\mathbb{L}$ itself. Furthermore, the rank decisions may suffer from badly conditioned problems, e.g. if the time scale is extremely small.

Nevertheless, the user can require the code to verify the given characteristic values after consistent initial values have been computed or after the BDF-solver successfully completed an iteration. In this case DGENDA returns an error message if any changes in the characteristic values are detected. The code also returns a suggestion for the correct characteristic values.

This feature may be used to compute these values by setting the parameter MXINDX sufficiently high (it must be at least equal to μ) and by supplying sufficient derivatives of the DAE.

Before any rank decisions are made during the computation, the matrix $[-N_\mu \;\; M_\mu]$ is equilibrated to lower its condition number. The code computes row and column scaling vectors s_r and s_c (stored in the arrays SCALC and SCALR), such that

$$[-N_\mu \;\; M_\mu] = \mathrm{diag}(s_r)^{-1}[-\bar{N}_\mu \;\; \bar{M}_\mu] \; \mathrm{diag}(s_c)^{-1} \; , \tag{15}$$

where the i-th element of s_c is an estimate for the highest absolute value of the i-th component of the solution approximation computed so far, and s_r is chosen such that the highest absolute value of every row of $[-\bar{N}_\mu \;\; \bar{M}_\mu]$ is equal to one. After computing the projectors $\tilde{Z}_1$, $\tilde{T}_2$ and $\tilde{Z}_2$, a back transformation is made. Note that these transformed projectors are no longer orthogonal but still project onto the appropriate subspaces.

The same scaling method is applied to the Jacobian (12) before the correction $\Delta\bar{x}_i$ is computed, to minimize the numerical errors caused by a badly conditioned problem.

Optionally the user can supply a scaling subroutine USCAL if an appropriate scaling method adapted to a certain problem is known. On the other hand it is also possible to deactivate any scaling.

10 Descriptor Systems

A descriptor systems is of the form

$$\begin{aligned} E(t)\dot{x}(t) = &\quad A(t)x(t) + B(t)u(t) + f(t) \\ y(t) = &\quad C(t)x(t) + D(t)u(t) + g(t). \end{aligned} \tag{16}$$

The mappings are as follows:

$$\begin{array}{ccccccccccccccc} x & : & \mathbb{R} & \to & \mathbb{R}^n & u & : & \mathbb{R} & \to & \mathbb{R}^m & y & : & \mathbb{R} & \to & \mathbb{R}^p \\ E, A & : & \mathbb{R} & \to & \mathbb{R}^{l\times n} & B & : & \mathbb{R} & \to & \mathbb{R}^{l\times m} & f & : & \mathbb{R} & \to & \mathbb{R}^l \\ C & : & \mathbb{R} & \to & \mathbb{R}^{p\times n} & D & : & \mathbb{R} & \to & \mathbb{R}^{p\times m} & g & : & \mathbb{R} & \to & \mathbb{R}^p \end{array}$$

The software package **GELDS** which is an enhancement of **GELDA** can be used to solve such a system. For this the system (16) is transformed into the equivalent linear DAE

$$\mathcal{E}(t)\dot{z}(t) = \mathcal{A}(t)z(t) + \mathcal{F}(t), \tag{17}$$

where

$$z = \begin{bmatrix} x \\ u \\ y \end{bmatrix}, \mathcal{E} = \begin{bmatrix} E & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathcal{A} = \begin{bmatrix} A & B & 0 \\ C & D & -I \end{bmatrix}, \mathcal{F} = \begin{bmatrix} f \\ g \end{bmatrix}.$$

If one wants to solve this problem one has to supply the functions

$$E, A, B, C, D, f \text{ and } g.$$

They have to be either usable for symbolic differentiation or according to the standards described in section 6. Apart from the fact that one has to provide seven functions (instead of three) the usage of the descriptor system solver is the same as the usage of the linear DAE solver.

First one has to call the setup function **dssetup** which opens a GUI where the user can adapt the solver to the considered problem. In fact, this GUI equals the GUI of **GELDA**. If needed, the next step would be to call **symdiff** to symbolically differentiate the given functions. Finally, the function **solvedae** is used to compute the solution.

- **dssetup**
  initializes a structure array that contains setup information for the descriptor system solver

  Syntax

  `setup = dssetup(l, n, m, p, tspan)`

  `s = dssetup(l, n, m, p, tspan, setting)`

  Arguments

  The following table describes the input arguments to the function geldasetup.

l	number of rows of E
n	number of state variables
m	number of input variables
p	number of output variables
tspan	1×2 array specifying t_0 and t_f
setting	string specifying the setting which is loaded 'standard': standard setup which is also loaded by default if *setting* is omitted 'large': setup for large systems

  The following table lists the output arguments for the function dssetup.

setup	structure array containing all information about the setup of the solver

Description

`s = dssetup(l, n, m, p, tspan, [setting])` opens a GUI where the user can edit the setup for the solver **dssolve**. Necessary information about the setup is stored in the structure array `s`.

- **symdiff**
  symbolically differentiates functions (see section 2)

Syntax

```
[darray] = symdiff(s, eh, ah, bh, ch, dh, fh, gh, ...
    'name1', 'name2', 'name3', 'name4', 'name5', 'name6', 'name7')
```

Arguments

The following table describes the input arguments to the function symdiff.

s	structure array for descriptor system solver
eh	handle to function $E(t)$
ah	handle to function $A(t)$
bh	handle to function $B(t)$
ch	handle to function $C(t)$
dh	handle to function $D(t)$
fh	handle to function $f(t)$
gh	handle to function $g(t)$
name1	name of the function implementing E and its derivatives
name2	name of the function implementing A and its derivatives
name3	name of the function implementing B and its derivatives
name4	name of the function implementing C and its derivatives
name5	name of the function implementing D and its derivatives
name6	name of the function implementing f and its derivatives
name7	name of the function implementing g and its derivatives

The following table lists the output arguments for the function symdiff.

darray	cell array containing handles to the generated functions

Description

The function symdiff symbolically differentiates the given functions and writes the results to files called `name?.m` in the current directory. Additionally, a cell array is returned containing handles to the functions implemented in these files. This cell array is necessary to run the descriptor system solver.

- **solvedae**

solves a descriptor system

Syntax

```
[T, X, xprime, cval, output, civ] = ...
    solvedae(s, darray, x0)

[T, X, xprime, cval, output, civ] = ...
    solvedae(s, darray, x0, tspan)
```

Arguments

The following table describes the input arguments to the function solvedae.

s	structure array for the descriptor system solver
darray	cell array containing handles to the functions E, A, B, C, D, f and g (in this order)
x0	not necessarily consistent initial values
tspan	time points where the solution will be evaluated

The following table lists the output arguments for the function solvedae.

T	vector containing the time points where the solution is evaluated if $tspan$ is not given $T = [t_0,\ t_f]$ else $T = tspan$
X	matrix containing the computed solution in which the ith row corresponds to the ith component of x and the jth column corresponds to the jth time point of T
xprime	vector which contains the first derivative of x at time t_f
cval	4×1 array containing the characteristic values of the descriptor system in the following order $cval(1) =$ strangeness-index $cval(2) =$ number of differential components $cval(3) =$ number of algebraic components $cval(4) =$ number of undetermined components
output	structure array (see section 7)
civ	consistent initial values

Description

The function **solvedae** calls the solver **dssolve** which transforms the descriptor system into a linear DAE and solves this new system. If one does not want to use symbolic differentiation one has to create a *cell array* containing function handles. Consult the MATLAB help to learn how to do so.

A Functions

name	description
acj2m	symbolically generates the Jacobians
adiffun	evaluates function $A \in \mathbb{R}^{l \times n}$
bdiffun	evaluates function $B \in \mathbb{R}^{l \times m}$
cdiffun	evaluates function $C \in \mathbb{R}^{p \times n}$
daefun	evaluates function F
daejac	evaluates a Jacobian of F
daesca	evaluates the user defined function for scaling purposes
ddiffun	evaluates function $D \in \mathbb{R}^{p \times m}$
dsprepare	supports symbolic differentiation for descriptor systems
dssetup	manages the GUI for **GELDS**
dssolve	solves descriptor system using **GELDS**
dummyuscal	is used if no user supplied scaling routine is given
ediffun	evaluates function $E \in \mathbb{R}^{l \times n}$
fdiffun	evaluates function f
gdiffun	evaluates function g
geldagui	implements the GUI for **GELDA/GELDS**
geldasetup	manages the GUI for **GELDA**
gendagui	implements the GUI for **GENDA**
gendasetup	manages the GUI for **GENDA**
ghd4civ	creates a separate file for the highest desired derivative to be used by function **fsolve** for correcting guessed consistent initial values
jac2f	writes Jacobians of F into a file

Table 3: All Functions of the Toolbox

name	description
lindaesolve	solves linear DAEs using **GELDA**
linerrmsg	displays error messages which occurred during the usage of **GELDA**
linprepare	supports symbolical differentiation for linear problems
loadlargesetting	loads setup for large linear problems
loadmbsetup	loads setup for simulating multi-body systems
loadstandardsetting	loads standard setup for linear problems
loadstandardsetup	loads standard setup for nonlinear problems
nldaesolve	solves nonlinear DAEs using **GENDA**
nlerrmsg	displays error messages which occurred during the usage of **GELDA**
nlprepare	supports symbolical differentiation for nonlinear problems
outof	checks whether given input equals certain numbers
propervector	checks vectors
propernumber	checks numbers
savesetting	saves structure array for linear problems
savesetup	saves structure array for nonlinear problems
smad2m	symbolically differentiates functions for nonlinear problems
smadlin2m	symbolically differentiates functions for linear problems
solvedae	picks right solver for given problem
supply	manages storage of input that cannot be given via the GUI
symdiff	starts symbolical differentiation
w2f	writes derivatives of F to file

Table 4: All Functions of the Toolbox continued

name	description
gelda	MEX-file for **GELDA**
genda	MEX-file for **GENDA**
gelds	MEX-file for **GELDS**
compile	this script compiles the toolbox

Table 5: MEX-Files and Scripts

References

[1] A. Ben-Israel and T. N. E. Greville. *Generalized Inverses: Theory and Applications*. John Wiley, New York, 1973.

[2] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential Algebraic Equations*, volume 14 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, 1996.

[3] P. Deuflhard. *Newton Methods for Nonlinear Problems*, volume 35 of *Springer series in computational mathematics*. Springer-Verlag, Berlin, 2004.

[4] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II.* Springer-Verlag, Berlin, 1991.

[5] P. Kunkel and V. Mehrmann. Canonical forms for linear differential-algebraic equations with variable coefficients. *J. Comput. Appl. Math.*, 56:225–259, 1994.

[6] P. Kunkel and V. Mehrmann. Regular solutions of nonlinear differential-algebraic equations and their numerical determination. *Numer. Math.*, 79:581–600, 1998.

[7] P. Kunkel, V. Mehrmann, W. Rath, and J. Weickert. GELDA: A software package for the solution of general linear differential algebraic equations. *SIAM J. Sci. Comput.*, 18:115 – 138, 1997.

[8] P. Kunkel, V. Mehrmann, and I. Seufer. GENDA: A software package for the numerical solution of general nonlinear differential-algebraic equations. Report, Technische Universität Berlin, Straße des 17. Juni 136, D-10623 Berlin, Germany, 2002.

[9] U. Nowak and L. Weimann. A family of Newton codes for systems of highly nonlinear equations - algorithm, implementation, application. Report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustraße 7, D-14195 Berlin, Germany, 1990.

[10] J. Ortega and W. Rheinboldt. *Iterative Solutions of Nonlinear Equations in Several Variables*, volume 30 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, 2000.

[11] L. R. Petzold. A description of DASSL: A differential/algebraic system solver. In R. S. Stepleman et al., editors, *IMACS Trans. Scientific Computing Vol. 1*, pages 65–68. North-Holland, Amsterdam, 1983.