DASSLC

User's Manual

Version 3.2 19/Oct/2007

Argimiro R. Secchi

GIMSCOP (Group of Integration, Modeling,
Simulation, Control, and Optimization of Processes)
Universidade Federal do Rio Grande do Sul
Departamento de Engenharia Química
Rua Eng. Luiz Englert s/n
90040-040 - Porto Alegre, RS - Brasil
arge@enq.ufrgs.br – http://www.enq.ufrgs.br

Revision and Copyright Information

The version 3 can deal with high-index DAEs by supplying a differential index vector of the dependent variables. The version 2 includes band matrix type to build the iteration matrix, makes few changes in the root structure, and, as major improvement, adds an iterative method to solve linear systems arising from Newton's iterations. Because the iterative method needs a preconditioner routine, this version is not totally downward compatible with earlier versions: it has a new last argument in the dasslc() call, and also a new last argument in the user-defined jacobian routine. The revisions up to 2.4 make some minor adjustments. The revision 2.5 includes a totally user supplied algebra package, including the iteration matrix building and storage. The daSetup() does not need an inputfile by setting inputfile="?" and the user may initialize the parameters in ROOT structure by hand, through a pointer to the user_init() function provided by the user (if not used then it must point to a NULL). The revision 3.0 included the differential index vector of the variables in daSetup() arguments to treat high-index problems. The revisions 3.1 and 3.2 created the function arSetOption() to set the solver options externally (after calling daSetup()) and removed the obligation to manually remove the rank information from the inputfile when set through daSetup() call. Also the default value of the absolute tolerance was changed to 1×10^{-10}.

The software development was sponsored in part by the National Council of Research and Development in Brazil, CNPq process numbers 351470/94-8, 300543/2003-4, and 307665/2006-2, in part by Research Support Foundation of the State of Rio Grande do Sul, FAPERGS process number 97/60093.93, in part by Brazilian Research and Development Financing Agency, FINEP process number 01.040.656.00, and in part by the Simulation Laboratory (LASIM) at UFRGS.

Contents

1 Tutorial

Introduction

DASSLC *(Differential-Algebraic System Solver in C)* is an easy-to-use and application independent C-based DAE solver package. It does the multirate integration of systems of differential-algebraic equations (DAEs). The integration algorithm used in DASSLC is an extension of the DASSL code of Linda R. Petzold [5]. The setup algorithm used in DASSLC is based on the DAWRS code [7, 6], a package to solve DAEs on parallel machines. Actually, DASSLC was initially extracted from DAWRS to run on sequential machines.

In order to help simulation problems, DASSLC also provides a simple Newton-Raphson's like algorithm to solve a fixed point problem (a steady-state solution).

This chapter provides a basic introduction, with a simple example, to use this software. Advanced options and set of commands are introduced in subsequent chapters.

Purpose

The scope of this program is to solve numerically an initial-value problem (IVP) in a system of differential-algebraic equations of the implicit form

$$\begin{aligned} F(t, y, \dot{y}, u) &= 0 \\ y(t_0) &= y_0 \\ \dot{y}(t_0) &= \dot{y}_0 \end{aligned} \tag{1.1}$$

where F: $R \times R^N \times R^N \times R^r \to R^N$ is a nonlinear function, $y(t) \in R^N$ is the vector of unknown variables, $\dot{y}(t) \in R^N$ is the vector of time derivatives of the unknown variables, and $u(t) \in R^r$ is the input vector. Systems of ordinary differential equations (ODEs) are included in this formulation as a special case of DAEs.

DASSLC uses the backward-differentiation-formula (BDF), with a fixed leading coefficient and variable stepsize and order, for the solution of DAEs of index zero and one [1]. DASSLC can also solve high-index DAEs when the user provides the differential index of each dependent variable (see the classical pendulum problem in Cartesian coordinates in the sample file `pend.c`). The integration algorithm used in DASSLC is an extension of the DASSL code [5].

Background

DASSLC, like most of the DAE/ODE solvers, is to be used as a subroutine. Thus, the user has to write the simulator layer, or the main program, and the set of equations to be solved. Therefore, some knowledge of C language is required in order to use this solver.

Some familiarity with the theory of DAEs is helpful, but not necessary, when solving high-index problems. The initial condition procedure is robust for low-index systems only, therefore it is necessary to provide a consistent initial guess to start a simulation when solving high-index problems. A more suitable algorithm to find a consistent initial condition is subject to the version 4.0. A better alternative to use DASSLC to solve high-index DAEs is through the simulation package EMSO [9] (http://www.enq.ufrgs.br/alsoc), which provides robust solvers for finding consistent initial conditions.

Installation

This code was already tested on DECstations, SUNs, IBM-PC compatible computers running posix or win32 operating systems, and on Cray's supercomputers.

As any subroutine, to use DASSLC, the source file has to be compiled with any C compiler, and linked with the user program and appropriated libraries. If the sparse linear algebra option is selected, then a sparse library has to be linked together.

There exist three variables: SPARSE, UNIX and CRAY, used to direct the compiler to generate a suitable object code. These variables are defined in the dasslc.h header file, and must be commented or uncommented according to the case.

The sparse linear system solver used in DASSLC is Sparse1.3 [4]. To install this solver, the source code should be placed in an appropriated directory, and its `Makefile` has to be executed as follow:

```
make sparse.a
```

Then, the library created, sparse.a, should be moved to a corresponding library directory. There is a specific make file, named Make_pc, to be executed on IBM-PC compatible computers instead of Makefile.

Any other sparse package can be used, the only requirement is the compatibility with the calling function arguments.

Demo Example

The usage of DASSLC is quite similar to that of most DAE/ODE solvers, specially DASSL. Values for y and $\dot{y}$ at the initial time must be given as input, and this values must be consistent, that is, if $t, y, \dot{y}$ are the given initial values, they must satisfy $F(t, y, \dot{y}, u) = 0$.

The example given here is a simple linear system of ODEs [3] described by the residual equations

$$\dot{y}_0 - d\,y_0 - y_1/e = 0 \tag{1.2}$$
$$\dot{y}_1 + y_1/e = 0 \tag{1.3}$$

where $d = -0.01$ and $e = 0.01$, which can be coded as follow to integrate the system from $t = 0$ to $t = 1$ with results every 0.1 units of time. The initial derivatives $\dot{y}(0)$ are assumed unknown. This example also computes the steady-state solution for syntax purpose only.

```
/*
 * $Log:          demo.c,v $
 * Revision 3.0  2007/07/12  10:29  arge
 * DASSLC version
 *
 */

#include "dasslc.h"

DASSLC_RES residuals;
DASSLC_JAC jacobian;
DASSLC_PSOL psolver;

void
main (int argc, char **argv)
{
  PTR_ROOT root;
  BOOL error;
  char *inputfile;
  REAL t = 0., tout = 0.1;

  if (argc > 1) inputfile = argv[1];
  else inputfile = "demo.dat";

  if (error = daSetup (inputfile, &root, residuals, 0, t, NULL, NULL, NULL, NULL,
    NULL, error < 0))
    {
      printf ("Setup error = %d\n", error);
      exit (1);
    }

  if (error = dasslc (INITIAL_COND, &root, residuals, &t, tout, jacobian, psolver),
     error < 0)
    printf ("error = %d\n", error);
  else
    for (; tout <= 1.; tout += .1)
        if (error = dasslc (TRANSIENT, &root, residuals, &t, tout, jacobian, psolver),
           error < 0)
          {
            printf ("error = %d\n", error);
            break;
          }

  daStat (root.savefile, &root);

  root.iter.atol[0] = 1e-20;
  if (!root.iter.stol) root.iter.atol[1] = 1e-20;
  if (error >= 0)
    if (error = dasslc (STEADY_STATE, &root, residuals, &t, tout, jacobian, psolver),
        error < 0)
      printf ("error = %d\n", error);

  daStat (root.savefile, &root);
  daFree (&root);
}                                           /* main */


BOOL
residuals (PTR_ROOT *root, REAL t, REAL *y, REAL *yp, REAL *res, BOOL *jac)
{
 BOOL error = FALSE;
 FAST int i, k;
 int rank, *index;
 PRIV REAL d = -.01, e = .01;

 if (*jac)
   {
     rank = root -> jac.rank;
     index = root -> jac.index;
   }
 else rank = root -> rank;

 for (k = 0; k < rank; k++)
     {
       i = (*jac ? index[k] : k);
       switch (i)
               {
                 case 0: res[i] = yp[0] - d * y[0] - y[1]/e; break;
                 case 1: res[i] = yp[1] + y[1]/e; break;
                 default: error = -1;
               }
     }

 return (error);
}                                           /* residuals */

#define PD(i, j) (*(pd + rank * (i) + j))

BOOL
jacobian (PTR_ROOT *root, REAL t, REAL *y, REAL *yp, REAL cj, void *ja,
            DASSLC_RES *residuals)
{
int rank = root -> rank;
```

```
REAL *pd = (REAL *)ja;
PRIV REAL d = -.01, e = .01;

PD(0,0) = cj - d;
PD(0,1) = -1.0/e;
PD(1,1) = cj + 1.0/e;

return FALSE;
}                                  /* jacobian */

/* preconditioner:  P x = b, where b also returns x */
BOOL
psolver (PTR_ROOT *root, REAL *b, DASSLC_RES *residuals)
{
PRIV REAL d = -.01, e = .01;
REAL cj = root -> iter.cj, c = 1./(1. + e * cj);

b[0] = (b[0] + b[1] * c)/(cj - d);
b[1]* = e * c;

return FALSE;
}                                  /* psolver */
```

The daSetup() function does all computations prior to the simulation (datafile readings, parameter settings, dynamic memory allocation, matrix structure determination, ...). The *INITIAL_COND* call of dasslc() finds the initial $\dot{y}$ if not given. Each *TRANSIENT* call of dasslc() solves the system from t to *tout*. The solution continues to get results at additional *tout* until reaches $t_f = 1$. The *STEADY_STATE* call dasslc() finds a steady-state solution for the given problem by setting all time derivatives to zero and solving the resulting fixed point problem. The daStat() function generate timing and counting data from the simulation and the daFree() deallocate all dynamic allocated memory.

The argument *jac is a flag to indicate what kind of call to residuals() was made. If *jac is TRUE ($\neq 0$) then residuals() evaluates the non-zero entries to the numerical jacobian formation, otherwise it computes all residuals. A simpler way to write the same residuals function, but without saving time to evaluate the iteration matrix, could be as follow:

```
BOOL
residuals (PTR_ROOT *root, REAL t, REAL *y, REAL *yp, REAL *res, BOOL *jac)
```

```
{
 PRIV REAL d = -.01, e = .01;

 res[0] = yp[0] - d * y[0] - y[1]/e;
 res[1] = yp[1] + y[1]/e;

 return FALSE;
}                                    /* residuals */
```

The jacobian() function is the user-defined dense iteration matrix, used when the option mtype userdense is defined. The psolver() function is the user-defined preconditioner to be used within the iterative linear algebra solver when the option linearmode iterative is defined. The argument *ja in the jacobian() function is a pointer to the iteration matrix. When using dense or band linear algebra it must be converted to a (REAL *) pointer, and when using sparse linear algebra to (**char** *) pointer (Sparse1.3) or any other pointer depending on the sparse package being used. In Chapter 2 we describe the residuals function and the iteration matrix in more detail.

The **if** condition in the main() function involving inputfile checks if the data are to be read from the demo.dat file or from the file given in the command line. The input file, demo.dat, used in this example has the form below.

```
#
# $Log:          demo.dat,v $
# Revision 2.0  97/05/29  11:40  arge
# DASSLC version
#

rank 2
option savefile demo.save
option !istall !stol
option maxlen 3000
option maxorder 5
print

data initial
time 0.
0: 1.
1: 1.
endata
```

```
#debug matrix

# some default options
#option mtype dense
#option sparsemode eval
#option linearmode direct
#option dampi 1.0 damps 1.0
```

The symbol # starts a comment until the end of line. The keyword **option** changes a set of default parameters, for example,

option maxorder 5

sets the maximum BDF order to five. To specify the initial condition and other sets of data, the environment **data** offers a variety of options. In this example

data initial
⋮
endata

gives the initial values of t and y as

$$t = 0$$
$$y_0(0) = 1$$
$$y_1(0) = 1$$

Note that $\dot{y}_0(0)$ and $\dot{y}_1(0)$ are not given. When this happen the initial condition has to be evaluated using $\dot{y}_0(0) = 0$ and $\dot{y}_1(0) = 0$ as initial guesses. Obviously, a better initial guess can be given in the input file.

The keyword **rank** specifies the system size. In Chapter 4 we describe all setting options. But, to conclude the demo example, the option

option savefile demo.save

tells DASSLC to save the simulation results in the demo.save file. The following save file results from a running of the main file described above on a PC486 microcomputer. Note that all default and changed parameters are listed in the file.

```
*** DASSLC - Differential/Algebraic System Solver in C ***
```

```
Copyright (C) 1992,2007 Argimiro R. Secchi, UFRGS - Version 3.0

Input file: demo.dat   Date: 12-07-2007    Time:   10:29:09

Iteration parameters:
 mtype dense
 linearmode direct
 sparsemode eval
 sparsethr 20
 nonblank 0.2
 maxnewton 4
 maxjacfix 5
 maxlen 3000

Convergence parameters:
 maxorder 5
 stepsize 0
 maxstep 0
 factor TRUE
 damps 1
 dampi 1
 convtol 0.33
 stol FALSE
 rtol (default) 1e-08
 atol (default) 0
 nonneg FALSE
 maxconvfail 10
 maxerrorfail 3
 jacrate 0.6
 maxsingular 3

Transient analysis parameters:
 differential index 0 or 1
 iststop FALSE
 istall FALSE

Debugging parameters:
 print TRUE
 newton FALSE
 bdf FALSE
 conv FALSE
 matrix FALSE

t =  0.000e+00  index      value    derivative   index      value    derivative
               (    0)  1.00000e+00  9.999e+01  (    1)  1.00000e+00 -1.000e+02

t =  1.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.99806e+00 -1.544e-02  (    1)  4.54000e-05 -4.540e-03
```

```
t =  2.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.99610e+00 -1.996e-02  (    1)  2.06116e-09 -2.061e-07

t =  3.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.99411e+00 -1.994e-02  (    1)  9.35765e-14 -9.358e-12

t =  4.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.99212e+00 -1.992e-02  (    1)  4.24837e-18 -4.248e-16

t =  5.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.99012e+00 -1.990e-02  (    1)  1.92876e-22 -1.929e-20

t =  6.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.98814e+00 -1.988e-02  (    1)  8.75655e-27 -8.757e-25

t =  7.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.98615e+00 -1.986e-02  (    1)  3.97547e-31 -3.975e-29

t =  8.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.98416e+00 -1.984e-02  (    1)  1.80486e-35 -1.805e-33

t =  9.000e-01  index      value    derivative   index      value    derivative
               (    0)  1.98218e+00 -1.982e-02  (    1)  8.19407e-40 -8.194e-38

t =  1.000e+00  index      value    derivative   index      value    derivative
               (    0)  1.98020e+00 -1.980e-02  (    1)  3.72011e-44 -3.720e-42

Timing information (in seconds): last mode Transient
    setup                  0.00
    steady state           0.00
    initial condition      0.00
    transient              0.49
    sparse structure       0.00
    perturb. matrix        0.00
    saving timepoints      0.00
    total time             0.49

Total number of time-points: 2155
Total number of evaluation of residuals: 2197
Total number of evaluation of jacobian: 28
Total number of Newton-Raphson iterations: 2197
Total number of error test failures: 9
Total number of convergence test failures: 0
Total number of rejected time-points: 9
Total number of rejected Newton-Raphson iterations: 10

Roundoff: 1.084202172485504430e-19
Total CPU-time (Setup + Simulation): 0.49 sec
Simulation time: 0.49 sec
```

```
t =  infinity   index      value    derivative   index      value    derivative
               (    0)  4.48416e-44  0.000e+00  (    1)  0.00000e+00  0.000e+00

Timing information (in seconds): last mode Steady-state
    setup                   0.00
    steady state            0.00
    initial condition       0.00
    transient               0.49
    sparse structure        0.00
    perturb. matrix         0.00
    saving timepoints       0.00
    total time              0.49

Total number of time-points: 2155
Total number of evaluation of residuals: 2201
Total number of evaluation of jacobian: 30
Total number of Newton-Raphson iterations: 2201
Total number of error test failures: 9
Total number of convergence test failures: 1
Total number of rejected time-points: 9
Total number of rejected Newton-Raphson iterations: 11

Roundoff: 1.0842021724855044 30e-19
Total CPU-time (Setup + Simulation): 0.49 sec
Simulation time: 0.49 sec
```

This file is divided into three parts:

- parameters settings
- simulation results
- timing and counting analyses

where the first part shows all simulation parameters. Those parameters that were not changed by the input file assume the default values. The second part shows those simulation results specified by the command **print** in the input file. If nothing is specified, then nothing is saved. The last part of the save file shows in detail the execution time for each program's phase and iteration countings.

2 Data Structure

Going back to the demo example (in Chapter 1), we can see that there are some non-standard C-declarations (PTR_ROOT, ROW_LIST, REAL, BOOL, etc.). Some of those declarations were defined to give more flexibility to the code. It is the case of

REAL → **double**
BOOL → **signed char**
SET → **signed char**
SHORT → **short**
PRIV → **static**
FAST → **register**

where, for example, REAL could be changed to **float** for single precision arithmetic. Other declarations can be simple or complex structures which contain specific information for each part of the algorithm. All these declarations can be found in the header file dasslc.h. Most of the declarations are transparent to the user. It is convenient to know some of them, like the ones listed above. However, there are three important structures that the user should have some knowledge of, at least part of these structures:

PTR_ROOT
JACOBIAN
DATABASE

which contain some variables and parameters that the user may use in his code. Therefore, here we only describe these three structures.

PTR_ROOT Structure

The PTR_ROOT structure is the basic structure for the DASSLC' subroutines. It contains those necessary structures for most of the functions, so that their arguments consist of this structure and only a few more parameters.

```
 typedef struct ptr_root       /* Root for all global pointers */
{
 FILE *savefile;               /* output file */
 char *filename;               /* output file name */
 char *pertfile;               /* perturbation matrix read/write file name */
 char alloc;                   /* bit set to allocated vars:  y,yp,prob,sub */
 SET mode;                     /* analysis mode */
 BOOL print;                   /* save and print solution if set */
 int *idxprint;                /* index of saved and printed variables */
 int rank;                     /* number of dependent variables */
 int *index;                   /* differential index of dependent variables */
 REAL t;                       /* independent variable (time) */
 REAL *y;                      /* vector of unknown variables */
 REAL *yp;                     /* vector of time derivatives */
 REAL *res;                    /* current residual vector */
 REAL *scratch;                /* scratch area of size = 2 * rank */
 void *user;                   /* pointer to user exchange data between routines */
 struct database *problem;     /* database problem pointer */
 struct database **sub_prob;   /* array of database sub-problem pointers */
 struct jacobian jac;          /* jacobian matrix structure */
 struct krylov kry;            /* krylov structure */
 struct bdf_data bdf;          /* BDF variables */
 struct iter_set iter;         /* iteration control parameters */
 struct debug_set debug;       /* debugging mode flag */
} PTR_ROOT;                    /* ptr_root */
```

The elements of the PTR_ROOT that the user should know are only

```
FILE *savefile
int rank
int *index
REAL t
REAL *y
REAL *yp
REAL *res
void *user
struct database *problem
struct database **sub_prob
```

which may be necessary in the main program and residuals, jacobian, and preconditioner functions. The last three elements are described in the following sections.

Looking at the demo example in Chapter 1, we can see in the daSetup()

function an argument named root, which is declared as PTR_ROOT. This argument is mandatory to this function, hence the user must declare a PTR_ROOT variable to initialize DASSLC. Also, the residuals(), jacobian(), and psolver() functions have a first PTR_ROOT argument to make the elements *rank*, **res*, **user*, **problem*, and ***sub_prob*, among others, accessible to the user inside of these functions.

The variables t, $y[rank]$, $yp[rank]$ in the PTR_ROOT structure of the main program are where DASSLC puts the solution at the end of a time interval, where *rank* is the dimension of the overall system of DAEs. When referred in the jacobian() and psolver() functions, the vector $res[rank]$ contains the current values of the residual vector. The $index[rank]$ is the differential index vector supplied by the user through the proper argument of the daSetup() for high-index problems. The *savefile* is the FILE pointer of the output file, where the results and other information are saved. The user can also use this file to save data. The file is opened in the daSetup() call and closed in the daFree() call.

JACOBIAN Structure

The following generic JACOBIAN structure

```
  typedef struct jacobian      /* Structure of iteration matrix */
{
  SET mtype;                   /* type of iteration matrix evaluation */
  char *matrix;                /* sparse/full matrix of partial derivatives */
  SHORT lband;                 /* # of diagonals below main diagonal */
  SHORT uband;                 /* # of diagonals above main diagonal */
  int *pivot;                  /* array of pivoting index */
  int rank;                    /* dimension of the local vector */
  int *index;                  /* local index of the active residuals */
  struct sparse_list *spl;     /* list of data dependency */
} JACOBIAN;                    /* jacobian */
```

where the structure SPARSE_LIST is

```
  typedef struct sparse_list   /* Structure of sparse data dependency */
{
  int size;                    /* row size */
  int *index;                /* column index vector */
} SPARSE_LIST;               /* sparse */
```

was designed to accommodate several iteration matrix special structures. Currently, only eight types of iteration matrix evaluations are possible:

- none evaluation
- band finite-difference iteration matrix
- sparse finite-difference iteration matrix
- dense finite-difference iteration matrix
- user-defined dense iteration matrix
- user-defined sparse iteration matrix
- user-defined band iteration matrix
- user-provided algebra package

Where the first type is only possible within the iterative linear algebra solver, and could be understood as a direct substitution method to solve the non-linear algebraic system. The parameter *mtype* specifies to the algorithm which one of the above types is to be used for the system.

In Chapter 3 we describe how the user may formulate the jacobian function for the dense, sparse, and band cases. In case of the evaluation by finite-differences, all related types follow the same rule: column perturbation, where only those non-zero elements are computed if the **jac* flag is used inside the residuals function (see the two ways to write residuals() function in the demo example, Chapter 1). For each column of the iteration matrix, the residuals function is called with $root -> jac.rank$ equal to the number of non-zero entries and $root -> jac.index[rank]$ contains the indices of the corresponding residual equations. To know when the residuals function is being called to form the iteration matrix, there exists a flag, named **jac*, in its arguments that is set to 1 ($TRUE$) in this case. With this flag the user has to select the correct number of residual evaluations (*rank*), and the correct equation indices.

The SPARSE_LIST is the list of all columns of the iteration matrix with their respective sizes (number of non-zero elements). The differences among the dense, sparse, and band cases are in the matrix structure and in the solver used to solve the linear system, but not in the manner the matrix is formed. The pointer *$matrix$ is declared as a **char** to generalize the matrix structure. When the dense and band cases are used this pointer is converted to a REAL pointer.

DATABASE Structure

Although each residual equation may have its own database, it is possible that some database be common to the overall system. Therefore we have to distinguish between the individual and the global database. It is done by the DATABASE **problem* and the DATABASE ***sub_prob* structures in PTR_ROOT. The DATABASE **problem* structure contains the global database, and the DATABASE ***sub_prob* structure contains the individual database. The ***sub_prob* is a ** pointer (or pointer of pointers) because each equation has its own database. This was the easiest way found to define the individual database without confusing the user. Thus, the **sub_prob*[i] corresponds to the database of the i-th residual equation.

A general and simple DATABASE structure was designed in DASSLC as follow.

```
  typedef struct database     /* lowest-level database structure */
{
  int nchar;                  /* number of characters */
  int nshort;
  int nint;
  int nlong;
  int nfloat;
  int ndouble;
  char *pchar;                /* pointer to characters */
  short *pshort;
  int *pint;
  long *plong;
  float *pfloat;
  double *pdouble;
} DATABASE;                   /* database */
```

This structure uses the basic C declarations that a set of data can have. The user only has to separate the type of data in common blocks, that is, integers with integers, reals with reals, etc., and to specify how many elements are in each group. In Chapter 4 we show how to write a database file.

Another way to exchange data among routines is let to be defined by the user. It can be done by means of the **void** **user* pointer inside the PTR_ROOT structure, which can be converted to any other type or structure.

3 Using DASSLC

In Chapter 1 we showed a simple example of how to use DASSLC. Here we describe a standard way to write the user's code for simple simulation. More complex simulations could be extended from it within the DASSLC' limitations.

Residuals Function

The residuals function can have any name, because it is passed as argument in the dasslc() call. This function must return an error condition, declared as BOOL. The error conditions can be: $TRUE$ (1) if the error should be bypassed in the next calls with different stepsizes; $FALSE$ (0) if no error was found; $ERROR$ (-1) if the problem cannot be solved (in this case the control is returned to the user); other conditions will be considered as $error = TRUE$.

For the given values of t, y and yp (time derivative $\dot{y}$), the subroutine should return in res the residual of the differential-algebraic system

$$res = F(t, y, yp)$$

where res, y and yp are vectors of length ($root -> rank$), i.e., the dimension of the system. The vector of pointers ($root -> sub\_prob$) has the addresses of the user-supplied database with individual indices of the residual equations, the pointer ($root -> problem$) has the user-supplied global database, and the pointer ($root -> user$) has the user-defined type of data.

To minimize the residual evaluation during the jacobian formation there is a flag $*jac$ to indicate if the call is for the numerical jacobian ($*jac = TRUE$) or if it is a call for the residual evaluation ($*jac = FALSE$). In the case $*jac = TRUE$ the $root$ structure gives a vector ($root -> jac.index$) of indices of the active residuals with length ($root -> jac.rank$). If the user set $*jac$ to $FALSE$, during the jacobian formation, then all ($root -> rank$) residuals have to be evaluated and the $index$ vector cannot be used. NOTE: never set $*jac = TRUE$.

The user must provide a subroutine of the form (or compatible):

```
#include "dasslc.h"

BOOL
residuals (PTR_ROOT *root, REAL t, REAL *y, REAL *yp, REAL *res, BOOL *jac)
{
  BOOL error = FALSE;
  FAST int i, k;
  int rank, *index;

  if (*jac)
    {
      rank = root -> jac.rank;
      local = root -> jac.index;
    }
  else rank = root -> rank;
                              ..................
  for (k = 0; k < rank; k++)
      {
        i = (*jac ? index[k] : k);
        switch (i)
                {
                  case 0: res[i] = $f_0(t, y, yp)$; break;
                  ...
                  case j: res[i] = $f_j(t, y, yp)$; break;
                  ...
                  case n: res[i] = $f_n(t, y, yp)$; break;
                  default: error = −1;
                }
      }

  return (error);
}
```

where $n = root -> rank - 1$. As C does not allow that n in the case statement be a variable, the user should find his way to define it.

Without using **jac* flag to minimize the residual evaluation during the jacobian formation, the residuals function has the form:

```
#include "dasslc.h"
```

```
BOOL
residuals (PTR_ROOT *$root$, REAL $t$, REAL *$y$, REAL *$yp$, REAL *$res$, BOOL *$jac$)
{
 BOOL $error = FALSE$;

 $res[0] = f_0(t, y, yp)$;
 ...
 $res[j] = f_j(t, y, yp)$;
 ...
 $res[n] = f_n(t, y, yp)$;

 return ($error$);
}
```

The residuals function must not alter y or yp.

Jacobian Function

The jacobian function can have any name, because it is passed as argument in the dasslc() call. The function must return an error condition, declared as BOOL. The error conditions can be: $TRUE$ (1) if the error should be bypassed in the next calls whit different stepsize; $FALSE$ (0) if no error was found; $ERROR$ (-1) if the problem cannot be solved (in this case the control is returned to the user); other conditions will be considered as $error = TRUE$.

For the given values of t, y and yp (time derivative $\dot{y}$), the subroutine should return the iteration matrix of the differential-algebraic system

$$F(t, y, yp) = 0 \Rightarrow pd = \frac{\partial F}{\partial y} + cj \frac{\partial F}{\partial yp}$$

where pd is a REAL matrix of dimension ($root -> rank \times root -> rank$) for the dense case and ($root -> rank \times (2 * root -> jac.lband + root -> jac.uband + 1)$), and a **char** pointer to the sparse structure for the sparse case, y and yp are vectors of length ($root -> rank$), i.e., the dimension of the system, and cj is a constant which changes whenever the stepsize or order of the BDF integration changes. The vector of pointers ($root -> sub\_prob$) has the addresses of the user-supplied database with individual indices of the residual equations, the pointer ($root -> problem$) has the user-supplied global database, and the pointer ($root -> user$) has the user-defined type of data.

If the user has set to numerical evaluation, then he/she can ignore this function. Otherwise, he/she must provide a subroutine, e.g., of the form

```
#include "dasslc.h"
#define J(i,j)            (*(pd + rank * (i) + j))

BOOL jacobian (PTR_ROOT *root, REAL t, REAL *y, REAL *yp, REAL cj, void *ja,
               DASSLC_RES *residuals)
{
  BOOL error = FALSE;
  REAL *pd = (REAL *)ja;
                         .......................

  J(0,0) = f_00(cj,t,y,yp)
  ...
  J(i,j) = f_ij(cj,t,y,yp);
  ...
  J(n,n) = f_nn(cj,t,y,yp);

  return (error);
}
```

where $n = root -> rank - 1$.

For the given values of t, y, yp, the subroutine must evaluate the non-zero partial derivatives for each equation and each variable, and store these values in the matrix *pd*. The elements of *pd* are set to zero before each call to the jacobian function, hence only non-zero elements need to be defined.

The way the user must store the elements into the *pd* matrix depends on the structure of the matrix which was indicated by the parameter *mtype*.

mtype = userdense:

when evaluating the (non-zero) partial derivative of i-th equation with respect to j-th variable, it must be stored in *pd* according to

$$J(i,j) = \frac{\partial F[i]}{\partial y[j]} + cj\frac{\partial F[i]}{\partial yp[j]}$$

mtype = usersparse:

the type of **pd* must be changed to the type of the sparse matrix structure pointer instead of REAL, and the entries must to be consistent with the sparse solver package.

For example, to use the Sparse1.3 solver [4], the jacobian function should be like:

```
#include "dasslc.h"

BOOL jacobian (PTR_ROOT *root, REAL t, REAL *y, REAL *yp, REAL cj, void *ja,
                DASSLC_RES *residuals)
{
 char *pd = (char *)ja;
 FAST int i, j;

 ...
 daSparse_value (pd, i, j) = f_ij(cj,t,y,yp);
 ...
}
```

where $J(i, j)$ in the previous subroutine is replaced by the function daSparse_value().

mtype = userband:

when evaluating the (non-zero) partial derivative of i-th equation with respect to j-th variable, it must be stored in *pd* according to

$$J(k, j) = \frac{\partial F[i]}{\partial y[j]} + cj\frac{\partial F[i]}{\partial yp[j]}$$

where $k = i - j + m$ and $m = root -> jac.lband + root -> jac.uband$. That means, the columns of the full matrix are stored in the columns of *pd* and the diagonals of the full matrix are stored in rows $(root -> jac.lband)$ through $(root -> jac.lband + m)$ of *pd*.

mtype = useralgebra:

the user must provide all routines to deal with the iteration matrix. In this case the jacobian function also has to factorize the iteration matrix, if needed by the user-provided linear algebra package. The user must also provide pointers to the functions: ujacFactor(), ujacSolve(), ujacPrint(), and ujacFree(), before calling daSetup(). In this mode, the user probably will have to allocate the matrix (when having special structure) in the setup phase, then it may be necessary to give a pointer to the user_init() function and call daSetup() with inputfile set to "?". See the example file daDemo2.c for more details.

In all cases, the jacobian function must not alter y or yp.

WARNING: earlier versions (1.*) do not have the residuals function as argument of the jacobian function.

Preconditionning Function

The preconditionning function can have any name, because it is passed as argument in the dasslc() call. The function must return an error condition, declared as BOOL. The error conditions can be: $TRUE$ (1) if the error should be bypassed in the next calls whit different stepsize; $FALSE$ (0) if no error was found; $ERROR$ (-1) if the problem cannot be solved (in this case the control is returned to the user); other conditions will be considered as $error = TRUE$. It is used only in the iterative method to solve the linear algebraic system. (see [2] for detail). If no preconditioner is given, that is, a NULL value is passed as argument of dasslc(), then a default preconditionning routine, using LU factorization with back- and forward substitution, is called to solve the system $Px = b$, and in this case P is the iteration matrix (or part of it). Naturally, if P is the full iteration matrix, then the default preconditioner is the worst situation (better to use the direct method).

A generic form of the preconditionning routine, where the iteration matrix type is not sparse, looks like (see psolver() in the demo example in Chapter 1):

```
#include "dasslc.h"
#define J(i,j)          (*(pd + rank * (i) + j))

BOOL psolver (PTR_ROOT *root, REAL *b, DASSLC_RES *residuals)
{
  BOOL $error = FALSE$;
  REAL *$pd$ = (REAL *)$root$ -> $jac.matrix$, *$cj$ = $root$ -> $iter.cj$;
  REAL *$res$ = $root$ -> $res$, $t$ = $root$ -> $t$, *$y$ = $root$ -> $y$, *$yp$ = $root$ -> $yp$;

                              .......................

  $b[0] = f_0(b, pd, cj, res, t, y, yp)$
  ...
  $b[i] = f_i(b, pd, cj, res, t, y, yp)$;
  ...
  $b[n] = f_n(b, pd, cj, res, t, y, yp)$;

  return (error);
}
```

where $n = root -> rank - 1$.

WARNING: earlier versions (1.*) do not have the preconditionning function as argument of dasslc().

Calls to DASSLC

In order to use DASSLC the user has to make calls to the following functions:

daSetup (*inputfile*, &*root*, *residuals*, *rank*, *t*, *y*, *yp*, *index*, *problem*, *sub_prob*)

dasslc (*STEADY_STATE*, &*root*, *residuals*, &*t*, &*tout*, *jacobian*, *psolver*)

dasslc (*INITIAL_COND*, &*root*, *residuals*, &*t*, &*tout*, *jacobian*, *psolver*)

dasslc (*TRANSIENT*, &*root*, *residuals*, &*t*, &*tout*, *jacobian*, *psolver*)

daStat (*root.savefile*, &*root*)

daFree (*root*)

where the arguments have to be declared as

char **inputfile*;
PTR_ROOT *root*;
int *rank*, **index*, *nbrk*;
REAL **y*, **yp*, *t*, *tout*;
DATABASE **problem*, ***sub_prob*;
BOOL residuals (PTR_ROOT *, REAL, REAL *, REAL *, REAL *, BOOL *);
BOOL jacobian (PTR_ROOT *, REAL, REAL *, REAL *, REAL, **void** *, DASSLC_RES *);
BOOL psolver (PTR_ROOT *, REAL *, DASSLC_RES *);

The daSetup() function is essential and must be the first call prior to dasslc(). This call reads and interprets all information in the *inputfile*, and does all computations prior to the simulation (parameter settings, dynamic memory allocation, ...) putting the results in the *root* structure. If *rank* (size of the system of DAEs) is zero, then it must be given in the *inputfile* as its first command (see Chapter 4). The next three arguments (*t*, *y*, and *yp*) are the initial condition. If a *NULL* pointer is given for the *y* argument, then the initial condition must also be in the *inputfile*, overwriting the ones given by argument. If *yp* is not set in any place, then daSetup() allocates space for it and puts zeros to all initial derivatives. The argument *index* is a user-supplied vector of size *rank* containing the differential index of the dependent variables. For index-0 and index-1 DAE systems this argument may be set to *NULL*. The files pend.c contains the classical pendulum problem in Cartesian coordinates to ilustrate the use of dasslc() to solve high-index DAE systems. The last two arguments, representing the global and individual database, can be *NULL* if there is no database or if they are given in the *inputfile*. If the user does not want use an inputfile, then the first argument must be set to "?". In this case the user may initialize the parameters in the ROOT structure by hand, through a pointer to the user_init() function provided by the user (if not used then it must point to a NULL before calling daSetup()). See files demo2.c and daDemo2.c that have examples not using inputfiles.

The *STEADY_STATE* call computes a steady state of the given system, using *y* as an initial guess. The values of *t* and *tout* are used only to direct the finite-difference evaluation of the Jacobian. The *INITIAL_COND* call finds the initial time derivatives, *yp*, if not given. The *TRANSIENT* call performs the transient analysis of the dynamic system from *t* to *tout*. At end of the simulation *t* shall have the value of *tout* to be used in the next call. The results of these calls are placed in *root.t* and the corresponding positions of the *root.y* and *root.yp* vectors.

The daStat() function generate timing and counting data from the simulation, saving the results in the specified file in the first argument. The daFree() function deallocates all dynamic allocated memory. It is possible to have more than one PTR_ROOT structure to solve different problems. Also, after a daFree() call, the same PTR_ROOT variable can be used to recall daSetup() to solve another problem.

Finally, to use the DASSLC, the user must include the following header file:

#include "dasslc.h"

which contains all necessary definitions of the DASSLC's structures and types, and defines all exit conditions of the code.

Error Messages

All calls to the dasslc() function return an error condition, that can be one of the following conditions:

- $INTERMED$ (1): a step was successfully taken in the intermediate-output mode; the code has not yet reached tout.
- $EXACT$ (2): the integration to tout was successfully completed ($t =$ tout) by stepping exactly to tout.
- $INTERPOL$ (3): the integration to tout was successfully completed ($t =$ tout) by stepping past tout. The values and their time derivatives, $y, \dot{y}$, are obtained by interpolation.
- $UNKNOWN$ (-1): unknown error
- $MAX\_TIMEPNT$ (-2): number of steps is too big. The maximum number of timepoints, specified by maxlen has been reached.
- INV_TOL (-3): error tolerance is too stringent. In this case the code does not return the control to the user, it increases the values of rtol and atol appropriately, in order to continue the integration.
- INV_WT (-4): element of wt is or become zero. The local error test cannot be satisfied because the user specified a zero component in atol and the corresponding computed solution component is zero. Thus, a pure relative error test is impossible for this component.
- $ERROR\_FAIL$ (-5): error test failed repeatedly. A singularity in the solution may be present or the initial condition is not consistent.
- $CONV\_FAIL$ (-6): corrector could not converge after maxconvfail attempts. An inaccurate or ill-conditioned iteration matrix may be the problem.

- *SINGULAR* (-7): iteration matrix is singular after maxsingular attempts. Some of the equations may be redundant, or the solution does not exist or is not unique. It is also possible that the system has an index problem (see [1, 6]).
- *MAX_ERR_FAIL* (-8): non convergence due repeated error test failures, maxerrorfail times. It is possible that the system is ill-posed, and cannot be solved using this code. It is also possible that there may be a discontinuity or a singularity in the solution, or even an index problem.
- *FAIL* (-9): non convergence due repeated error in residuals. Multiple calls to residuals have been unsuccessfully made, with different stepsizes, to attempt to bypass a problem in the residuals function.
- *CALL_ERROR* (-10): incorrigible error in residuals function indicated by its returned error condition ($error = ERROR$).
- *INV_MTYPE* (-15): invalid matrix type to linear solver. Either the mtype = none was used with the direct method or a NULL value was passed as a user-defined jacobian function.
- *INV_TOUT* (-33): $t =$ tout
- *INV_RUN* (-34): last step was interrupted by an error
- *INV_RTOL* (-35): some element of rtol is negative
- *INV_ATOL* (-36): some element of atol is negative
- *ZERO_TOL* (-37): all elements of rtol and atol are zero
- *TOO_CLOSE* (-38): tout too close to t to start
- *TOUT_BEH_T* (-39): tout behind t
- *TSTOP_BEH_T* (-40): tstop behind t
- *TSTOP_BEH_TOUT* (-41): tstop behind tout

Note that the errors are reported only by negative values of the error condition. Values less than (-14) occurs when invalid input is detected.

4 Input File Options

Here we describe all the options and commands available for the DASSLC's input file. In Chapter 1 we already introduced an example of an input file used by the demo example. The setup data can be read either from an input file or from the standard input (usually defined to be the keyboard). It is very simple to write the data into the input file. There exists a set of commands followed by a set of options. A symbol # starts a comment, and from that point to the end of line everything is treated as a commentary. Blank lines, multiple spaces, and tab spaces are allowed, and have no effect in the data. Multiple options for a command can be used in a single line, but multiple commands are not allowed in one line, except if the separator symbol ; is used between commands. To use more than one line for multiple options the continuation symbol \ (backslash) has to be used at end of each line. The continuation symbol is not valid for a comment, but everything that follow a continuation symbol to the end of line are treated as commentary. The available commands for the input file are:

- **rank**
- **inputfile**
- **data**
- **option**
- **print**
- **debug**

Note: all commands, excepting **rank**, overwrite themselves, that is, if there is a similar command's option in the input file, then only the last one has effect.

rank

The **rank** command sets the size of the system of DAEs. This command must be the first command in the input file, and naturally only one **rank** command is allowed.

usage: **rank** *size*

The *size* of the system can be as big as possible (limited by total memory), and greater than zero.

If not present in the input file, the system size has to be passed as an argument by the daSetup() function in the user's program, see Chapter 3. If the system size is defined by the user's program, then the **rank** command is not necessary in the input file, but if present and different from the one provide by the user's program then daSetup() will return an error message.

inputfile

This command nests multiple input files. When DASSLC finds an **inputfile** command it reads the specified input file and executes its commands, and then returns to the previous input file. There is no default extension for the input filenames.

usage: **inputfile** filename

The filename may or may not be between quotes like: "filename", 'filename', ‘filename‘, or ‘filename'.

data

The **data** command is a structured command to read the sparse matrix structure, the parameter settings for a set of variables, the initial condition (or initial guess), and the user database. It starts with the keyword **data** and must end with an **endata** keyword.

usage: **data** option
⋮
endata

The available options for the **data** commands are:

- value
- residual
- database
- sparse
- initial

where each option has its own set of suboptions to describe the data structure.

value

The option value sets all individual information about the variables, and during the setup phase these information are processed to generate an appropriate set of parameters for the overall system.

usage: **data** value *list*
suboption
⋮
suboption
endata

The information given by the suboptions are set to the variables specified by the *list* of indices. The *list* may assume one of the following formats, or a combination of them.

```
value 0 1 3 4 6          (individuals)
value 0-3 5-9            (continuous intervals)
value 0-8,2              (step interval)
value                    (all variables)
```

A generic formatting of a list can be writing as:

[first[-last[,step]]]

where *[ ]* means optional.

The available suboptions are:

suboption	type	default	lower	upper	short description
nonneg	BOOL	0	0	1	nonnegative solution
rtol	REAL	$1e^{-8}$	0	$HUGE$	relative tolerance
atol	REAL	$1e^{-10}$	0	$HUGE$	absolute tolerance

where $HUGE$ represents the highest machine number of the respective type, *lower* is the lower bound and *upper* is the upper bound for a suboption. A simple example to use this option can be written as:

```
data value 0 3-5 7 9-15,3
 rtol 1e-7
 atol 1e-10
endata
```

The nonneg suboption, as all BOOL type options hereafter, can be used in two different ways:

nonneg 0 or !nonneg

for a $FALSE$ setting and

nonneg 1 or nonneg

for a $TRUE$ setting. This option is used when the user knows that the solution will (or will not) always be nonnegative. However, it is probably best to try the code using the nonneg option set to $FALSE$ first, and only to use it set to $TRUE$ if that does not work very well. The strategy to choose the final settings for the overall system is based on the following priority: if any variable is set to !nonneg then the final set will be !nonneg, where the settings for those variables not specified in the value *list* are taken as the global settings given by the **option** command described later.

The suboptions rtol and atol are the relative and absolute tolerances, respectively. Setting atol $= 0$ results in a pure relative error test on the specified variables, and setting rtol $= 0$ results in a pure absolute error test. A mixed test with non-zero rtol and atol corresponds roughly to a relative error test when the solution components are much bigger than atol and to an absolute error test when the solution components are 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 too much accuracy was requested and change to the maximum accuracy it believes possible. The priority is to use vectorial tolerances, that is, if any of these tolerances is set for some variables using the **data** value, then the corresponding variables will use these tolerances, where the remain variables will use the global tolerances (given by the **option** command).

residual

The option residual reads the user's database for each specified residual function according to the DATABASE structure, see Chapter 2.

```
usage: data residual list
        suboption
            ⋮
        suboption
       endata
```

The *list* format is similar to that described for the value option. Based on the DATABASE structure, the suboption can assume any one of the available types (**char**, **short**, **int**, **long**, **float**, **double**), according to the format

type size `data`

where **type** can by one of the above types, size is the number of `data` entries. For example,

```
data residual 1 4-7 10-18,2
```

```
 int 2 1 -1
 double 3 1.5e-6 -0.01 3.E-5
 char 31 "vapor" 'liquid' `regular` feed `ideal gas'
 double 10 .01 .11 .20 .015 .12 \
           .005 .14 .03 .17 .1
endata
```

defines the individual database for the specified list of equations. Note that here the index list is made up of equation indices, whereas in the value option the index list constitutes of indices of variables. The **char** type accepts entries between quotes or without quotes, where the quotes are useful to insert spaces in a single entry and to avoid the conversion of a numeric entry to integer (if an entry is a number, then it is converted to one-byte integer). The second **double** option uses the continuation symbol to utilize more than one line.

database

Similarly to the residual option, the database option reads the user's database according to the DATABASE structure, but in this case the data belong to the global database.

usage: **data** database
```
 suboption
     ⋮
 suboption
endata
```

Obviously, this option does not have an index list. The suboptions are the same ones used in the residual option. It is possible to have more than one **data** database command in the input file, as well as the **data** residual command. The database are not overwritten.

All data contained in the database option are stored in the DATABASE **problem* structure, which is part of the PTR_ROOT *root* structure. In case of the residual option, the data are stored in the DATABASE ***sub_prob* of the PTR_ROOT *root* structure. Another way to define the local and global databases is directly from the user program by the daSetup() function, see Chapter 3.

sparse

The sparse option offers two different ways to define the structure of the iteration matrix (or jacobian matrix for the steady-state): the variable index i structure and the absolute index structure. See also the perturbation file in the next section.

usage: **data** sparse
  i = *equation list* : [+/-] *operation list*
  ⋮
  i = *equation list* : [+/-] *operation list*
  equation list : *variable list*
  ⋮
  equation list : *variable list*
endata

where *equation* and *variable* represent the equation index and variable index, respectively. The *list* format is the same used in value option. The i structure has two basic index operations: addition (+) and subtraction (-). The plus signal is optional. It means that the equation i has the variables $i+$ *list* or $i-$ *list*. Each i = statement can have only one basic operation on the command line.

In order for DASSLC to make use of the user's sparse structure, it has to be activated. This is done by either one of the following commands, described in the subsequent section:

option sparsemode infile
option sparsemode eval

Examples of user's sparse structure can be written as follows.

```
option sparsemode infile
data sparse
 i = 0 3 1 4 :  0-5,2
 i = 0-9 :  -0-3
 1 3 5-9,2 :  0-9,3 2 8
 0-19 :  0-19
 i = 0-19 :  0-19
 i = :  0
endata
```

where the first structure tells that equation `0` has the variables: `0 2 4`, equation `3`: `3 5 7`, equation `1`: `1 3 5`, and equation `4` the variables `4 6 8`. The second structure means a 10×10 matrix with the main diagonal and three lower sub-diagonals. The third structure has the equations `1 3 5 7 9` with the variables `0 3 6 9 2 8`. The forth is a full matrix. The next one is an upper triangular matrix, and the last one is a diagonal matrix.

initial

In Chapter 3 we showed one form to define the initial condition or initial guess, where the values are passed by argument in the daSetup() function. Other possibility is using the structured command **data**.

usage: **data** initial
 time *t*
 list : *y yp*
 ⋮
 list : *y yp*
 endata

At least one *y* value or *t* must be given to avoid error messages. If *t* is not given, then it is assumed to be zero. Also, if some *y* or *yp* values are not given they are assumed to be zero. It is not possible to specify a *yp* value without specify it corresponding *y* value, because the order *y yp* must hold. The *list* format, representing the indices of the variables, is the same used in the value option. If more than one *list* is put in the same line, then the intermediate *yp* values must be present, except if the separator symbol ; is used between *lists*. The example,

```
data initial
 time 0.1
 0-6,2:  3.5
 1 3:  1.3 -5.7  5:  -2.1    # two lists in one line.
 7-8:  0.5;  9:  2.2 -5.0    # two lists with separator.
```
endata

sets $t = 0.1$, $y = \{3.5, 1.3, 3.5, 1.3, 3.5, -2.1, 3.5, 0.5, 0.5, 2.2\}$, and $yp = \{0.0, -5.7, 0.0, -5.7, 0.0, 0.0, 0.0, 0.0, 0.0, -5.0\}$. If the given initial condition is not consistent, then the user has to make the $INITIAL\_COND$ call with the dasslc() function. The initial condition algorithm used in DASSLC only computes the initial time derivatives for the given y values.

option

The **option** command is used to set most of the adjustable parameters in DASSLC. Similarly to the value option, the **option** command has a set of suboptions with their default values and valid range.

usage: **option** option *value*

The available options and their *value* default and range, for each type, are:

option (BOOL)	default	lower	upper	short description
factor	1	0	1	convergence factor activation
iststop	0	0	1	stop-time activation
istall	0	0	1	intermediate results
stol	1	0	1	scalar tolerances
nonneg	0	0	1	nonnegative solution
restart	1	0	1	restarting activation

When the factor option is $TRUE$ (1) DASSLC computes the convergence acceleration factor to be multiplied by the residuals, during the transient solution. For the direct method, this factor minimize the differences between the current iteration matrix (when not computed) and its approximation from previous iterations. For the iterative method it is just a heuristic acceleration factor, based on same ideas.

When the iststop option is $TRUE$ DASSLC checks for stopping points. The iststop option can also be changed at any time during the simulation by changing the value of $root.iter.iststop$ and $root.iter.tstop$.

The istall option returns the control to the user after each evaluated timepoint, and they are saved in the save file if required by a **print** command. The !istall option returns the control only if $tout$ was reached or if some error has occurred. Depending on the stol option the tolerances rtol and atol can be either scalars ($TRUE$) or vectors ($FALSE$).

A nonneg option implies a nonnegative solution. Note that this and other options are equivalent to the suboptions of the value option. The reason for it is that the user may want a different parameter setting for some variables. Thus, the options given by the **option** command are valid for all variables, except for those specified in the value's index list with an equivalent suboption.

The restart option, when set to $TRUE$, activates the restarting strategy to solve the linear system iteratively by the Krylov's method, and ($root -> kry.maxrest$) indicates the maximum number of restarts.

option (SHORT)	default	lower	upper	short description
maxorder	5	1	20	maximum BDF order
lband	-1	-1	$HUGE$	# of diagonals below main diagonal
uband	-1	-1	$HUGE$	# of diagonals above main diagonal
maxl	5	1	20	max # of iterations before restart
kmp	5	1	20	# of orthogonalized vectors, $\leq$ maxl
maxrest	5	1	$HUGE$	max # of restarts
sparsethr	20	2	$HUGE$	threshold for sparse matrix usage
maxconvfail	10	0	$HUGE$	max # of convergence test failures
maxerrorfail	3	0	$HUGE$	max # of local error test failures
maxsingular	3	0	$HUGE$	max # of singular iteration matrices
maxnewton	4	1	$HUGE$	max # of Newton iterations
maxjacfix	5	1	$HUGE$	max # of non-updated jacobians
maxlen	100	2	$HUGE$	max # of timepoints per interval

Although a BDF order greater than 6 may be unstable for DAEs [1], the upper bound for the maximum BDF order (maxorder) was fixed to 20 (which can be changed by the macro MAX_ORDER in the header file dasslc.h), but its default is 5. Some test examples were found to be more efficient with maximum BDF order set to 8. If storage is a severe problem, the user can save some locations by restricting maxorder (for each order decrease, the code requires $2\,(N{+}5)$ fewer locations), however it is likely to be slower.

When using band matrix type, the user must specify the number of diagonals below the main diagonal (lband) and the number of diagonals above the main diagonal (uband). For matrix types not defined by the user, a -1 value to lband and/or uband tells the code to find out their values, using the sparsemode selected. When $2 * lband + uband \geq rank$ a warning message is generated suggesting to use dense matrix. Even for nonnegative given values of lband and uband, DASSLC computes their values, based on the sparsemode, for comparative purpose only.

The parameters maxl, kmp, and maxrest are used by the Krylov's method in the iterative solution of linear systems [2]. When the restarting algorithm is activated by the option restart, maxl is the maximum number of linear iterations allowed before restarting, and maxrest is the maximum number of restarts. The parameter kmp ($\leq$ maxl) sets the number of vectors on which the orthogonalization is done. When kmp = maxl a complete generalized minimal residual (GMRES) iteration is performed, otherwise we have the incomplete form. The upper bound for maxl and kmp was fixed to 20 (which can be changed by the macro MAX_LI in the header file dasslc.h).

If the system size is less than sparsethr and the matrix type (mtype) was set to sparse, then the matrix type is changed to dense. The sparse linear system solver has an additional overhead to initialize and work with the sparse matrix structure, hence for low-order matrices (≤ 20) dense solvers are less expensive.

The testing failures parameters (maxconvfail – for convergence problem, or maxerrorfail – for large local error), the maximum number of singular iteration matrices (maxsingular) and Newton-Raphson iteration (maxnewton) are taken per time-point. The maxnewton parameter is multiplied by the $STEADY\_FACTOR$ when finding a steady-state, which is set to be 5 in the header file dasslc.h, and by the $INIT\_FACTOR$ when evaluating the initial condition, set to 3 in dasslc.h. The maxjacfix parameter sets the maximum number of Newton-Raphson iterations without updating the iteration matrix during the steady-state and initial condition analyses.

If the maximum number of timesteps per interval (maxlen) is reached, then the control is returned to the user, so he/she can decide to stop the simulation or recall dasslc().

option (REAL)	default	lower	upper	short description
jacrate	0.6	0.01	1.0	range for do not update jacobian
nonblank	0.2	0.0	1.0	fraction of non-zero entries
damps	1.0	$1.e^{-3}$	1.0	SS damping factor in N-R iteration
dampi	1.0	$1.e^{-3}$	1.0	IC damping factor in N-R iteration
tstop	$HUGE$	$-HUGE$	$HUGE$	stop point
rtol	$1.e^{-8}$	0.0	$HUGE$	relative tolerance
atol	$1.e^{-10}$	0.0	$HUGE$	absolute tolerance
stepsize	0.0	$-HUGE$	$HUGE$	initial stepsize
maxstep	0.0	0.0	$HUGE$	maximum stepsize
convtol	0.33	$1.e^{-3}$	1.0	N-R conv. tolerance in w-norm
litol	0.05	0.0	$HUGE$	linear iterations tolerance

The jacrate threshold is used to decide whether a new iteration matrix is necessary during the transient analysis. Roughly, it sets the range of allowed change in the stepsize and BDF order without updating the iteration matrix, i.e.,

$$\mathsf{jacrate} < \frac{\alpha}{\hat{\alpha}} < \frac{1}{\mathsf{jacrate}}$$

where α is a constant which changes whenever the stepsize or order changes, and $\hat{\alpha}$ depends on the stepsize and order at some past time step when the last iteration matrix was computed [1].

If the iteration matrix structure has more than nonblank % non-zero entries and the matrix type (mtype) was set to sparse, then the matrix type is changed to dense.

The damps and dampi parameters are factors that multiplies the residuals in the Newton-Raphson iterations during the steady-state and initial condition analyses, respectively. The default values in earlier versions were set to 0.75, which are more conservative, however we noted that in many cases it is not necessary to damp the iterations. So, we let to the user decide to damp them.

For some problems it may not be permissible to integrate past a point tstop because a discontinuity occurs there, or the solution or its time derivative is not defined beyond tstop. When iststop is set to $TRUE$, then the code does not integrate past tstop. In this case any *tout* beyond tstop is not valid (see also the comments about the parameter iststop).

The suboptions rtol and atol are the relative and absolute tolerances, respectively. Setting atol $= 0$ results in a pure relative error test on the specified variables, and setting rtol $= 0$ results in a pure absolute error test. A mixed test with non-zero rtol and atol corresponds roughly to a relative error test when the solution components are much bigger than atol and to an absolute error test when the solution components are 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 too much accuracy was requested and change to the maximum accuracy it believes possible. This options are also available with the data value command, in order to have different tolerances for some variables. Thus, the options given by the **option** command are valid for all variables, except for those specified in the value's index list with an equivalent suboption.

Systems of differential-algebraic equations may occasionally suffer from severe scaling difficulties on the first step. It can be alleviated by specifying an initial stepsize, if the user knows a great deal about the scaling of the problem. A zero stepsize means an automatic stepsize computed by DASSLC, and a zero maxstep means no upper bound for the absolute value of the stepsize.

DASSLC uses the weighted norm

$$\|y\|_w = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y_i}{wt_i}\right)^2}$$

where wt is the weight vector, which depend on the error tolerances specified by rtol and atol (in a scalar or vector fashion to account for scaling of the problem), defined as

$$wt_i = rtol_i \, |Y_i| + atol_i$$

where Y is a representative vector of a vector sequence (e.g., it is the y_n vector during the time step from t_n to t_{n+1}). In this norm an iteration is taken to have converged when

$$\frac{\rho}{1-\rho}\|y^{(k)} - y^{(k-1)}\|_w \leq 1.0$$

where ρ is the convergence rate. In order to avoid that the errors due terminating the Newton-Raphson's iterations affect adversely the integration error estimates, an extra restriction factor (convtol) is added to the tolerance, that is

$$\frac{\rho}{1-\rho}\|y^{(k)} - y^{(k-1)}\|_w \leq \mathsf{convtol}$$

For example, if the weights are all equals and proportional to a desired tolerance ϵ, then the above condition may be interpreted as

$$\frac{\rho}{1-\rho}\|y^{(k)} - y^{(k-1)}\|_2 \leq \mathsf{convtol} \, . \, \epsilon$$

The iterative solution of linear system ($A\,x = b$) arising from a Newton-Raphson's iteration, using Krylov's method, is taken to have converged when

$$\|b - A\,x\|_w \leq \mathsf{litol}\,.\,\mathsf{convtol}$$

where litol is the tolerance for the linear iterations with an additional convtol factor, explained above.

Most of the default values for the parameters related to the integrator were based on similar parameters in DASSL, other ones were found to be more adequate. The upper and lower bounds were fixed in such a form to give a maximum flexibility to the user without degrading the code.

option (**char**)	default	short description
mtype	dense	iteration matrix type
linearmode	direct	linear solver mode
sparsemode	eval	sparse structure mode
savefile	stdout	save filename
inputpert	–	perturbation input filename
savepert	–	perturbation save filename

The iteration matrix types (mtype) available at moment are:

- none (0): iteration matrix not computed
- band (1): finite-difference band matrix
- sparse (2): finite-difference sparse matrix
- dense (3): finite-difference dense matrix
- userdense (4): dense matrix computed by the user's code
- usersparse (5): sparse matrix computed by the user's code
- userband (6): band matrix computed by the user's code
- useralgebra (7): user-provided algebra package

where the numbers between parenthesis are the identifiers inside the code, defined in the header file (dasslc.h). New types of iteration matrix can be implemented with the following rule for the identifiers: the number 0 is reserved to the none type; user-defined types must be placed above userdense; and automatic modes must be placed below dense.

If the user does not provide a subroutine to evaluate the iteration matrix analytically, it will be approximated by numerical differencing. The none type is allowed in the linearmode iterative only. It is useful when computing the preconditioner together with the iteration matrix is better than to compute them separately. When no preconditionning is carried out, the none type also can be viewed as a direct substitution method to solve the non-linear algebraic system.

Although it is less trouble for the user to allow the code compute partial derivatives by finite differences, the solution will be more reliable if the user provide the derivatives. Sometimes numerical differencing is cheaper than evaluating derivatives analytically and sometimes it is not – this depends on the problem being solved.

In the useralgebra type the user must provide all routines to deal with the iteration matrix. In this case the jacobian function also has to factorize the iteration matrix, if needed by the user-provided linear algebra package. The user must also provide pointers to the functions: ujacFactor(), ujacSolve(), ujacPrint(), and ujacFree(), before calling daSetup(). In this mode, the user probably will have to allocate the matrix (when having special structure) in the setup phase, then it may be necessary to give a pointer to the user_init() function and call daSetup() with inputfile set to "?". See the example file daDemo2.c for more details.

Excepting the filenames, all **char** options offer a set of possibilities that can be increased in further implementation of DASSLC. Also, the **char** option settings may be quoted or not.

The linear solver modes (linearmode) available to solve the linear system arising from a Newton-Raphson's iteration are:

- direct (0): direct method
- iterative (1): iterative method
- both (2): direct and iterative methods

The direct method uses LU factorization with back- and forward substitution to solve the linear system. The iterative method uses a Krylov-type method:

the scaled GMRES (Generalized Minimal Residual) with left preconditionning [2]. With the both solvers mode the user can select either direct or iterative modes during the dasslc() calls. It is the same as direct mode if no change in the variable (*root* −> *jac.mtype*) is made. The reason for this type is to save storage space when using the other types alone.

There are three sparse structure modes available:

- infile (0): user-defined sparse structure
- none (1): no sparse structure (dense or band matrix)
- eval (2): dependency matrix evaluation

where in the infile mode the user has to define all entries in the sparse matrix structure using the **data** sparse command. The eval mode finds all entries by means of perturbation over the unknown variables, symbolically represented by the *differential-algebraic dependency matrix* [8]. If sparsemode mode is set to eval and there are some user information in the **data** sparse command, then these information are used too.

The savefile option defines the name of the save file where all parameter settings, required simulation results, and timing and counting information are saved. The inputpert and savepert options define the name of the perturbation file where the available dependency matrix are read from and saved to, respectively. Usually, it is left to DASSLC to create the perturbation file to be used in subsequent simulations. However, it can also be created by the user. In this case the perturbation file must be a binary file with the following format:

[row size][dependency symbol][indices]

⋮

[row size][dependency symbol][indices]

where *row size* is the size of each row of the dependency matrix (note that the matrix can have a sparse structure), the *dependency symbol* is one of the following one-byte number for each D_{ij} entry in the i-th matrix row

0	$\rightarrow$	if the i-th equation does not involve y_j or $\dot{y}_j$
1	$\rightarrow$	if the i-th equation involves y_j but not $\dot{y}_j$
2	$\rightarrow$	if the i-th equation involves $\dot{y}_j$ but not y_j
3	$\rightarrow$	if the i-th equation involves both y_j and $\dot{y}_j$

the field *indices* contains the respective indices (declared as **int**) of each entry of the matrix row. The j-th column of the perturbation matrix is generated in DASSLC by

$$\left| \frac{F(t, y + \delta_j e_j, \dot{y} + \frac{\delta_j}{h} e_j, u) - F(t, y, \dot{y}, u)}{\delta_j} \right| \tag{4.1}$$

e_j is the j-th unit vector, δ_j is the perturbation, and h is a given stepsize.

print

The **print** command specifies which simulation results have to be saved in the save file additionally to the parameter settings and the timing and counting information.

usage: **print** *list*

If no **print** command is given, then nothing else is saved. The variables *list* format is the same used in the value option.

debug

The **debug** command sets debugging flags which are used to print partial simulation results according to the specified option.

usage: **debug** option

If no option is given, then all available options are set to $TRUE$. The available options are:

- newton: information about Newton-Raphson iterations
- bdf: print BDF data at begin of an interval
- conv: print convergence information
- matrix: print iteration matrix

5 Outputs

In Chapter 1 we showed an output resulting of a simulation of the demo example. This output is saved in the save file, if it is given (or in the `stdout` if not), and basically constitutes of three parts:

- parameter settings
- required simulation results
- timing and counting information

In the first part all default values or the changes made by the commands described in Chapter 4 are saved. The second part saves the required simulation results specified by the **print** command. Finally, if the function `daStat()` is called, then the third part saves the timing and counting information.

Other outputs can be obtained by setting the debug options with the **debug** command in the input file.

The output information are easily to understand hence they are not discussed here.

6 Interface to MATLAB

A mex implementation of DASSLC is also provided in the file dasslc2ml.c to solve DAEs inside MATLAB (http://www.mathworks.com/), which is based on the work of S. Hauan and S. Storen (2005) at Carnegie Mellon University. The mex file was successful compiled in version 5.2 of MATLAB, using the command:

mex -V4 -output dasslc dasslc2ml.c dasslc.c

and the compiled mex file run successful also in MATLAB 6.2. The full MATLAB command line to call DASSLC is given below.

[t,y,yp,outp] = dasslc(F,tspan,y0,rpar,rtol,atol,index,inputfile,jac)

where the input arguments (minimum of 3) are:

argument	**type**	**short description**
F	string	name of the residuals function
tspan	vector	[T0 T1 ... TFINAL]
y0	vector	initial condition
rpar	array	optional arguments transfered to function F
rtol	scalar	optional relative tolerance parameter (default: $1e^{-8}$)
atol	scalar	optional absolute tolerance parameter (default: $1e^{-10}$)
index	vector	optional differential index of each variable
inputfile	string	optional input file as described in this manual
jac	string	optional name of the user-provided jacobian function

and the outputs (minimum of 2) are:

argument	**type**	**short description**
t	vector	vector of independent variable
y	matrix	matrix of dependent variables (each line is a timestep)
yp	matrix	optional matrix of time-derivative of dependent variables
outp	scalar	optional solution flag, positive and 0 if successful

The residuals function provided by the user as a MATLAB M-file should have the following format:

function [res,ires]=dydt(t,y,yp,rpar)

where $ires = 0$ if $dydt$ has no errors, and $rpar$ is the optional argument transfered by the calling function dasslc(); res is the vector of the residuals of the DAE system; $dydt$ is any function name given by the user, usually the same name of the M-file.

The jacobian function provided by the user as a MATLAB M-file should have the following format:

function [M,ires]=jacmx(t,y,yp,cj,rpar)

where M is the transpose of the iteration matrix $\frac{\partial F}{\partial y} + cj \frac{\partial F}{\partial yp}$, cj is a constant provided by the integrator that changes whenever the stepsize or the order of the BDF changes, and the other parameters are the same as described in the residuals function. $jacmx$ is any function name given by the user, usually the same name of the M-file. Up to now, only full iteration matrix can be provided by the user via MATLAB.

The files test_pend.m, test.m, and test1.m are examples of DAEs solved by this interface.

References

[1] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problem in Differential-Algebraic Equations.* North-Holland, New York, 1989.

[2] P. N. Brown, A. C. Hindamarsh, and L. R. Petzold. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM J. Sci. Comput.*, pages 1467–1488, November 1994.

[3] K. Dekker and J. G. Verwer. *Stability of Runge-Kutta methods for stiff nonlinear differential equations.* Elsevier Science Publishers, New York, 1984.

[4] K. S. Kundert and A. L. S. Vincentelli. A sparse linear equation solver - user's guide. Technical report, Dept. of EE and CS, Univers. of California Berkeley, Berkeley, 1988.

[5] L. R. Petzold. DASSL: Differential algebraic system solver. Technical Report Category #D2A2, Sandia National Laboratories, Livermore, California, 1983.

[6] A. R. Secchi. *Simulação Dinâmica de Processos Químicos pelo Método da Relaxação em Forma de Onda em Computadores Paralelos.* PhD thesis, COPPE/UFRJ, Rio de Janeiro, RJ, October 1992.

[7] A. R. Secchi, M. Morari, and E. C. Biscaia Jr. Dawrs: A differential-algebraic system solver by the waveform relaxation method. In *Proceedings of The Sixth Distributed Memory Computing Conference (DMCC6)*, pages 502–505, Portland, Oregon, April 1991.

[8] A. R. Secchi, M. Morari, and E. C. Biscaia Jr. The waveform relaxation method for concurrent dynamic process simulation. In *Proceedings of AIChE Annual Meeting*, Los Angeles, California, November 1991.

[9] R. P. Soares and A. R. Secchi. EMSO: A new environment for modeling, simulation and optimization. In *Proceedings of ESCAPE* 13^{th} *European Symposium on Computer Aided Process Engineering*, pages 947–952, Lappeenranta, Finland, June 2003.