Journal of Computational and Applied Mathematics 236 (2012) 4119–4131

Contents lists available at SciVerse ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

A Test Set for stiff Initial Value Problem Solvers in the open source software R: Package **deTestSet**

Francesca Mazzia[a], Jeff R. Cash[b], Karline Soetaert[c,*]

[a] *Dipartimento di Matematica, Università degli Studi di Bari, Via Orabona 4, 70125 Bari, Italy*

[b] *Department of Mathematics, South Kensington Campus, Imperial College London, London SW7 2AZ, United Kingdom*

[c] *Royal Netherlands Institute of Sea Research (NIOZ), 4401 NT Yerseke, The Netherlands*

ARTICLE INFO

Article history:
Received 15 February 2012
Received in revised form 19 March 2012

Dedicated to the memory of Donato Trigiante

MSC:
primary 65L04
65L05
secondary 65L80

Keywords:
Ordinary differential equations
Differential algebraic equations
Implicit differential equations
Initial value problems
Test problems
R

ABSTRACT

In this paper we present the R package **deTestSet** that includes challenging test problems written as ordinary differential equations (ODEs), differential algebraic equations (DAEs) of index up to 3 and implicit differential equations (IDEs). In addition it includes 6 new codes to solve initial value problems (IVPs). The R package is derived from the Test Set for Initial Value Problem Solvers available at http://www.dm.uniba.it/~testset which includes documentation of the test problems, experimental results from a number of proven solvers, and Fortran subroutines providing a common interface to the defining problem functions. Many of these facilities are now available in the R package **deTestSet**, which comprises an R interface to the test problems and to most of the Fortran solvers. The package **deTestSet** is free software which is distributed under the GNU General Public License, as part of the R open source software project.

1. Introduction

This paper is dedicated to Donato Trigiante. Donato was a driving force behind the idea of setting up test sets for mathematical problems and greatly encouraged his students to pursue this aim. The idea to develop a Test Set for stiff Initial Value Problems was first discussed at the workshop *ODE to NODE*, held in Geiranger, Norway, 19–22 June 1995. It was felt that both engineers and computational scientists alike can benefit from having a standard test set for IVPs which includes documentation of the test problems, experimental results from a number of proven solvers and Fortran subroutines providing a common interface to the defining problem functions. Engineers are able to see at a glance which methods will potentially be the most effective for their class of problems. Researchers can compare their new methods with the results of existing ones without incurring additional programming workload. Peter van der Houwen and his students at CWI in Amsterdam were the first to develop and maintain a practical test set for Initial Value Problems [1]. From 2001 on, the project has been maintained by the INdAM Bari unit project group "Codes and test problems for Differential Equations" [2]. The Test

* Corresponding author.
E-mail addresses: mazzia@dm.uniba.it (F. Mazzia), j.cash@imperial.ac.uk (J.R. Cash), karline.soetaert@nioz.nl (K. Soetaert).
URLs: http://www.dm.uniba.it/~mazzia (F. Mazzia), http://www2.imperial.ac.uk/~jcash/ (J.R. Cash), http://www.nioo.knaw.nl/users/ksoetaert (K. Soetaert).

0377-0427/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2012.03.014

Set for Initial Value Problem Solvers (hereafter referred to as IVPTESTSET) is available at http://www.dm.uniba.it/~testset and includes documentation of the test problems, experimental results from a number of well-established solvers, and Fortran subroutines providing a common interface to the defining problem functions.

Apart from the engineers and scientists that are acquainted with working with compiled languages such as Fortran or C, there are many scientists who prefer to solve their problems in high-level problem solving environments (PSEs). Until now, the most often used PSEs for solving differential equations are commercial packages such as Matlab [3], Mathematica [4] or Maple [5], or open-source software such as Scilab [6] or Octave [7].

One of the emerging PSEs whose use is expanding rapidly, especially in universities and academia, is the open source software R [8]. Although still mainly known as software for visualization and statistics, R has recently been extended to also provide powerful methods for solving differential equations [9] by a number of extension packages. The R package **deSolve** [10] provides solution methods for initial value problems of ODEs, DDEs, PDEs and DAEs. The R packages **rootSolve** [11] and **ReacTran** [12] include more solution methods for PDEs, while **bvpSolve** [13] is designed for solving boundary value problems.

Here we describe a new R package **deTestSet** that adds some powerful methods for solving DAEs and provides an R interface to most of the test problems in the IVPTESTSET.

The main aim of this paper is to introduce the new package and to present a comparison of the performance of well-known solvers using challenging test problems in the R environment. The paper is structured as follows. First we list the main characteristics of the test set of initial value problems [2] and define the classes of problems that we are dealing with. In Section 2.1, the classes of the test problems are briefly discussed and some implementation issues noted. Section 3 introduces the integration routines available in the new package, while Section 4 gives some example implementations of ODE, DAE and IDE systems in R with numerical benchmarks of computational performance. Finally some concluding remarks are given in Section 5.

The package is available from the Comprehensive R Archive Network at http://CRAN.R-project.org/package=deTestSet. New versions of the package, still under development, are available at http://r-forge.r-project.org/ [14].

2. The test set for stiff initial value problems

The main characteristics of the IVPTESTSET [2] are as follows:

- uniform presentation of the problems,
- description of the origin of the problems,
- robust interfaces between problem and drivers,
- portability among different platforms,
- contributions by people from several application fields,
- presence of real-life problems,
- tested and debugged by a large, international group of researchers,
- comparisons with the performance of well-known solvers,
- interpretation of the numerical solution in terms of the application field,
- ease of access and use.

2.1. The test problems

The test problems in IVPTESTSET can be categorized into three classes: systems of Ordinary Differential Equations (ODEs), Differential Algebraic Equations (DAEs), Implicit Differential Equations (IDEs). In this test set we call a problem an ODE if it has the form

$$\begin{aligned} &y' = f(t, y), \quad t_0 \le t \le t_{end} \\ &y, f \in R^d \\ &y(t_0) \text{ is given.} \end{aligned} \tag{2.1}$$

A problem is called a DAE if it is of the form

$$\begin{aligned} &My' = f(t, y), \quad t_0 \le t \le t_{end} \\ &y, f \in R^d, \quad M \in R^{d\times d} \\ &y(t_0) \text{ is given} \end{aligned} \tag{2.2}$$

where M is a constant, possibly singular, matrix.

The label IDE is given to problems which can be cast immediately into the form

$$\begin{aligned} &f(t, y, y') = 0, \quad t_0 \le t \le t_{end} \\ &y, f \in R^d \\ &y(t_0) \quad \text{and} \quad y'(t_0) \text{ are given.} \end{aligned} \tag{2.3}$$

Note that ODEs and DAEs are subclasses of IDEs. In what follows we give a list of the ODE test problems available in **deTestSet**. A complete description of the problems is available in [2]

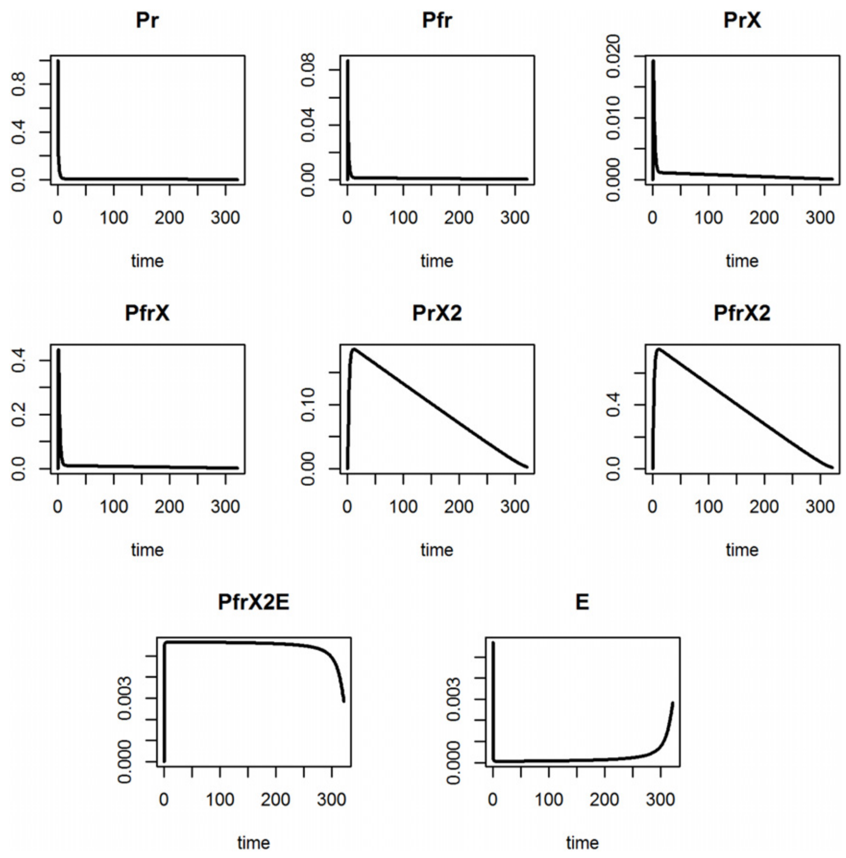

Fig. 1. Numerical solution of the HIRES problem. See text for the R code.

- ODE Problems: HIRES, Pollution, Ring Modulator, Medical Akzo Nobel, EMEP, Pleiades, Van der Pol, Robertson, Orego, Beam, E5.
- DAE Problems: Chemical Akzo Nobel, Andrews' squeezing mechanism, Transistor amplifier, Charge pump, Two bit adding unit, Car axis, Fekete, Slider crank, Water tube system.
- IDE Problems: NAND gate, Wheelset.

2.2. The R implementation of the test problems

For each problem we have implemented an `R` interface to the `Fortran` function available in the IVPTESTSET, suitably modified in order to be consistent with the **deSolve** package. To run, for example, the problem HIRES with the default solver (`mebdfi`, [15]) and with the solver `radau` [16] from the `R` package **deSolve**, we just need in `R` to open the package (`library`), and to call the `R` function `hires` with the following code:

```
library(deTestSet)
out  <- hires()
```

```
Problem HIRES
Solved with mebdfi
Using rtol = 1e-06, atol=1e-06
Mixed error significant digits:
5.567281
```

```
out2 <- hires(method = radau)
```

```
Problem HIRES
Solved with radau5
Using rtol = 1e-06, atol=1e-06
Mixed error significant digits:
9.103287
```

This solves the problem HIRES and gives as output the solution, printing the mixed error significant digits (see Section 2.3) computed with respect to a reference solution. To plot the solution we write

```
plot(out, lwd = 2)
```

This simple statement plots all 8 dependent variables constituting the HIRES problem at once, using a line width twice the size of the default (`lwd`) (see Fig. 1).

2.3. Mixed error significant digits

All the codes in **deSolve** and **deTestSet** implement an error estimate, but it is not assured that the error will be of the same order of magnitude as the prescribed tolerances, which by default are `1e-6`.

A common way to compare codes is to use the so-called work precision diagrams using the *mixed error significant digits*, *mescd*, defined by

$$mescd := -\log_{10}(\max(|absolute\ error/(atol/rtol + |ytrue|)|)), \tag{2.4}$$

where the *absolute error* is computed at all the mesh points at which output is wanted, *atol* and *rtol* are the input absolute and relative tolerances, *ytrue* is a more accurate solution computed using one of the available solvers with smaller relative and absolute input tolerances and where ($/$, $+$ and max) are element by element operators.

We will use this quantity for comparing the efficiency of the different codes.

3. The integration routines

The `R` package **deTestSet** is closely related to the package **deSolve** [10] from which it inherits the calling sequence for the test problems and the integration codes. Moreover, it includes six more codes to solve ODEs, DAEs and IDEs, which we introduce next.

3.1. Explicit Runge–Kutta solvers

Two explicit Runge–Kutta codes available in **deTestSet** are based on the well-established Fortran codes `dopri5` and `dop853` [17]. In addition, a new code based on the Cash–Karp Runge–Kutta method [18,19], has been implemented.

The general calling sequence for solving ODEs in `R` is

```
ode(y, times, func, parms, ...)
```

where `times` holds the times at which output is wanted, `y` holds the initial conditions, `func` is the `R` function that describes the differential equations, and `parms` contains the parameter values (or is `NULL`, i.e. undefined). Many additional inputs can be provided, e.g. the absolute and relative error tolerances (defaults `atol = 1e-6, rtol = 1e-6`), the maximal number of steps (`maxsteps`), the integration `method` used etc.

The default integration method for the function `ode` is `lsoda` from the **deSolve** package, which implements the type-insensitive solver `LSODA` [20].

To solve ODEs with any of the new methods we write

```
library(deTestSet)
ode(y, times, func, parms, method = cashkarp, ...)
ode(y, times, func, parms, method = dopri5, ...)
ode(y, times, func, parms, method = dop853, ...)
```

Alternatively, we can also call the new integrators directly:

```
cashkarp(y, times, func, parms, ...)
dopri5  (y, times, func, parms,...)
dop853  (y, times, func, parms,...)
```

If we type `?cashkarp` a help page that contains a list of all options that can be changed is opened. As most of these options have a default value, we are not obliged to assign a value to them, as long as we are content with the default.

3.2. Implicit solvers

Several solvers from the `R` package **deSolve** are based on implicit schemes, e.g. many solvers from `ODEPACK` [21], the code `VODE` [22], `DASPK20` [23] and the code `RADAU5` [16]. Of these only the last two can be used for the solution of DAE problems written in Hessenberg form of index at most 3.

There are three more codes based on implicit schemes available in **deTestSet**. The first one is based on modified extended backward differentiation formulas `mebdfi` [15], the second one on generalized Adams methods `gamd` [24] and the last is based on blended implicit methods `bimd` [25,26]. They all can solve ODEs and DAEs written in Hessenberg form of index up to 3 (for a definition of index see [9,16]). The code `mebdfi` can also solve IDEs. The calling sequence for solving ODEs with these methods is

```
library(deTestSet)
ode(y, times, func, parms, method = gamd, ...)
ode(y, times, func, parms, method = mebdfi, ...)
ode(y, times, func, parms, method = bimd, ...)
```

while for solving DAE problems it is

```
library(deTestSet)
dae(y, times, func, parms, method = gamd, mass, ...)
dae(y, times, func, parms, method = mebdfi, mass, ...)
dae(y, times, func, parms, method = bimd, mass, ...)
```

where argument `mass` is the matrix M (Eq. (2.2)). The solution of IDEs can be written as

```
mebdfi(y, times, parms, res, dy, ...)
```

where `res` is the R function that implements the residual equations (2.3) and `dy` holds the initial values of the derivatives.

These new functions can be used together with `daspk` [23] and `radau` [16] from **deSolve**. Only `daspk` and `mebdfi` can be used for the solution of IDEs in R, without transforming the problem.

3.3. Stiffness detection

The main characteristic of the new explicit Runge–Kutta solvers is that they all include at least one test for stiffness detection, and by default they stop if stiffness is detected. This feature explains why we included these solvers in a Test Set for *Stiff* Initial Value Problems. It is possible to continue even if stiffness is detected using the argument `verbose = TRUE` for `dopri5` and `dopri853` and setting the argument `stiffness=-2 or stiffness=-3` for `cashkarp`.

The code `cashkarp` includes two stiffness detection algorithms described in [16, p. 21]. The first method, originally proposed by Shampine [27] is based on comparing two error estimates of different orders $err = O(h^p)$, $\widetilde{err} = O(h^q)$, with $q < p$. Usually $err \ll \widetilde{err}$, if the step size is limited by accuracy requirements and $err > \widetilde{err}$ when the step size is limited by stability requirements. The precise way in which the two error estimators are derived is described in [17, p. 21]. This test is inserted in `cashkarp`. The practical test used is: if $0.1err > \widetilde{err}$ for at least 15 steps, separated by at most 6 steps where this inequality is not satisfied then the problem is taken to be stiff (parameter `stiffness=3`).

The second way to detect stiffness is to approximate the dominant eigenvalue of the Jacobian as follows. Let v denote an approximation to the eigenvector corresponding to the dominant eigenvalue of the Jacobian. If the norm of v ($\|v\|$) is sufficiently small then the Mean Value Theorem tells us that a good approximation to the dominant eigenvalue is

$$|\bar{\lambda}| = \frac{\|f(t, y+v) - f(t, y)\|}{\|v\|}. \tag{3.1}$$

The cost of this procedure is at most 2, but often 1, function evaluations per step. In the code `cashkarp` we use two different approximations of y_{n+1} at the point t_{n+1}, that are available since the fifth stage computes another approximation in $t_{n+1} = t_n + c_5 h$, therefore

$$|\bar{\lambda}| = \frac{\|f(t_{n+1}, y_{n+1}) - f(t_n + c_5 h, g_5)\|}{\|y_{n+1} - g_5\|}.$$

The product $h|\bar{\lambda}|$ can then be compared to the linear stability boundary of the method in order to detect if the stepsize is limited by stability.

The practical test used in the code `dopri5` is: if $h|\bar{\lambda}| > 3.25$ (6.1 for `dopri853`) for at least 15 steps, separated by at most 6 steps where this inequality is not satisfied then the problem is taken to be stiff. The test used in the code `cashkarp` is the same as in `dopri5`. This test is the default for all the solvers.

3.4. Implementation issues

In practice, the Fortran codes are implemented in R via a wrapper, written in C, that forms the interface between the Fortran and the R codes. The original codes have been modified in order to have the calling sequence of the derivative function consistent with the calling sequence in the **deSolve** package. Note that for the codes in **deSolve** the default maximum step size is made a function of the `times` argument (the time sequence where output is wanted), whereas for `gamd` and `bimd` the default maximum step size is set equal to $|t_{end} - t_0|$.

4. Numerical experiments

This section deals with the solution of problems included in **deTestSet**. Some of them are described in detail, so the users can learn how to solve a problem using R. For many of the examples taken from the IVPTESTSET we just describe the numerical results. The first example also introduces the techniques used for the benchmarking.

4.1. ODE problems

4.1.1. The van der Pol equation

A commonly used example to demonstrate stiffness is the van der Pol problem (see [16, p. 566]). It is defined by the following second order differential equation:

$$y'' - \mu(1-y^2)y' + y = 0, \tag{4.1}$$

where μ is a parameter. We convert (4.1) into a first order system of ODEs by adding an extra variable, representing the first order derivative:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1-y_1^2)y_2 - y_1. \end{aligned} \tag{4.2}$$

Stiff problems are obtained for large μ, non-stiff for small μ. We ran the model for $\mu = 1, 10, 1000$, using `mebdfi` as the integrator. The implementation in R is

```
yini  <- c(y = 2, dy = 0)
Vdpol <- function(t, y, mu)
    list(c(y[2],
           mu * (1 - y[1]^2) * y[2] - y[1]))
```

```
times <- seq(from = 0, to = 30, by = 1)
nonstiff <- mebdfi(func = Vdpol, parms = 1, y = yini,
                   times = times)
interm   <- mebdfi(func = Vdpol, parms = 10, y = yini,
                   times = times)
stiff    <- mebdfi(func = Vdpol, parms = 1000, y = yini,
                   times = 0 : 2000)
```

The function `diagnostics` prints important information about the numerical solution, such as the value of the output flag that informs us if the computation has been successful, the number of function evaluations and so on.

```
 diagnostics(nonstiff)
```

```
--------------------
mebdfi return code
--------------------

  return code (idid) =  0
  Integration was successful.

--------------------
INTEGER values
--------------------

  1 The return code : 0
  2 The number of steps taken for the problem so far: 657
  3 The number of function evaluations for the problem so far: 2472
  4 The number of Jacobian evaluations so far: 59
  5 The method order last used (successfully): 7
 10 The number of matrix LU decompositions so far: 59
 13 The number of error test failures of the integrator so far: 17
 18 The order (or maximum order) of the method: 7
 19 The number of backsolves so far 2294
 20 The number of times a new coefficient matrix has been formed so far 59
 21 The number of times the order of the method has been changed so far 6

--------------------
RSTATE values
--------------------

  1 The step size in t last used (successfully): 0.03067547
```

Using the parameter $\mu = 1$ the problem is not stiff, so we can apply an explicit Runge–Kutta solver, printing the time the solver takes (in seconds) to solve the problem:

```
system.time(nonstiff_cashkarp <- cashkarp(func = Vdpol, parms = 1,
                y = yini, times = times))
```

```
user  system elapsed
0.02    0.00    0.01
```

```
system.time(nonstiff_dopri5 <- dopri5(func = Vdpol, parms = 1,
                y = yini, times = times))
```

```
user  system elapsed
0.04    0.00    0.03
```

```
system.time(nonstiff_dopri853 <- dopri853(func = Vdpol, parms = 1,
                y = yini, times = times) )
```

```
user  system elapsed
0.03    0.00    0.03
```

They all compute the solution without experiencing any problems. However if we use $\mu = 100$, and the default tolerances, they detect stiffness and the output is, for example:

```
interm_cashkarp <- cashkarp(func = Vdpol, parms = 100, y = yini,
                times = times)
```

```
THE PROBLEM SEEMS TO BECOME STIFF AT X = 0.20488351681251302
 USING THE STANDARD CHECK OF EIGENVALUES
Warning messages:
1: In rk5(y, times, func, parms, rtol, atol, verbose, hmax,  :
  problem is probably stiff - interrupted
2: In rk5(y, times, func, parms, rtol, atol, verbose, hmax,  :
  Returning early. Results are accurate, as far as they go
```

Up until now we compared the execution time of the codes to solve the same problem, using the default relative and absolute tolerances.

To compare the solver's efficiencies, for every solver, a range of input tolerances was used to produce plots of the resulting `mescd` values against the number of CPU seconds needed for a run, when not otherwise specified the reference solution has been computed using `bimd` with $rtol = atol = 10^{-14}$. We took the average of the elapsed CPU times of 4 runs. The format of these diagrams is as in [17,16, pp. 166–167,324–325]. As an example we report in Fig. 2 the work precision diagrams for the methods `lsoda`, `dopri5`, `cashkarp`, `dopri853`, `adams`[1] (on the left) and the work precision diagrams for the methods `mebdfi`, `radau`, `gamd`, `bimd`, `daspk` (on the right) running the van der Pol problem with $\mu = 1$. The range of tolerances used is, for all codes, $rtol = 10^{-4-j(5/8)}$ with $j = 0, \ldots, 16$, all the other parameters are the default, except for `radau` we use `hmax` equal to the width of the integration interval. We use `times <- 0:30` for $\mu = 1$.

We wish to emphasize that the reader should be careful when using these diagrams for a mutual comparison of the solvers. The diagrams just show the result of runs with the prescribed input on the specified computer.

A more sophisticated setting of the input parameters, another computer or compiler, as well as another range of tolerances, or even another choice of the input vector `times` may change the diagrams considerably. We used a Personal Computer with Intel(R) Core(TM)2 Duo CPU (U9400 1.40 GHz, 2,80 GB of RAM) and MICROSOFT WINDOWS XP. We show the impact of the output times chosen by running the vdpol problem with $\mu = 1000$ using `times <- 0:2000` and `times <- seq(0, 2000, by = 100)`. The results are in Fig. 3.

For the van der Pol problem for $\mu = 1$ the `adams` is the most efficient code (Fig. 2), while for $\mu = 1000$ `bimd` require the least computational effort to compute a solution with a similar number of *mescd* (Fig. 3).

4.1.2. Pleiades problem

The Pleiades problem [17] is from celestial mechanics. This problem has been described in detail in the IVPTESTSET [2]. We report here the R code of the problem as it was implemented in [9], to compare the results from the implementation in pure R and from the R interface to the Fortran code. The problem involves seven stars, with masses m_i, in the two-dimensional plane of coordinates (x, y). The stars are considered to be point masses. The only force acting on them is gravitational attraction,

[1] The `adams` method is available from **deSolve**, and is based on the solver LSODE from ODEPACK [21].

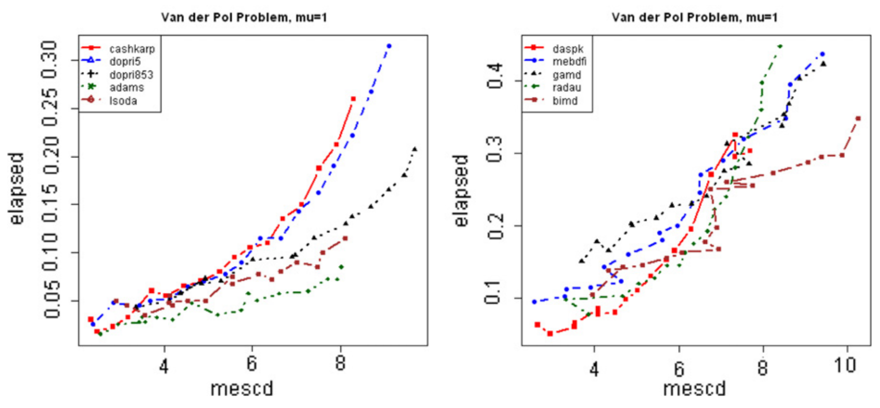

Fig. 2. Work precision diagrams for the van der Pol problem, $\mu = 1$.

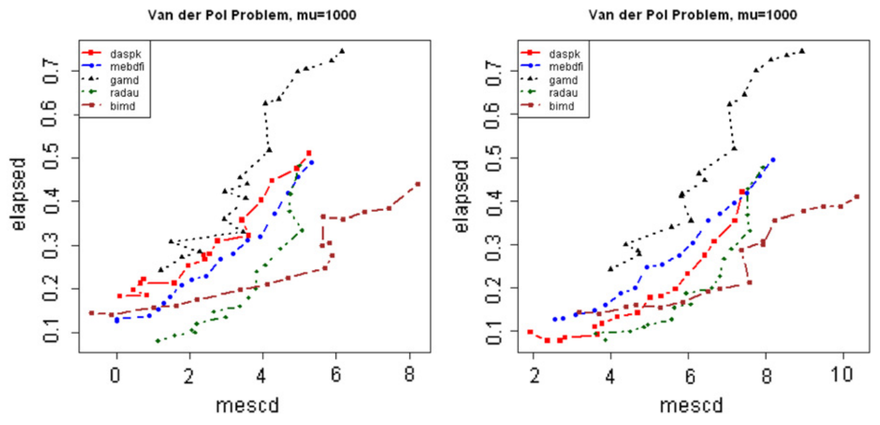

Fig. 3. Work precision diagrams for the van der Pol problem $\mu = 1000$. `times <- 0:2000` on the left, `times <- seq(0, 2000), by=100` on the right.

with gravitational constant G (units of $\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$). If $r_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2$ is the square of the distance between stars i and j, then the second order equations describing their movement are given by

$$\begin{aligned} x_i'' &= G \sum_{j \neq i} m_j \frac{(x_j - x_i)}{r_{ij}^{3/2}} \\ y_i'' &= G \sum_{j \neq i} m_j \frac{(y_j - y_i)}{r_{ij}^{3/2}}, \end{aligned} \tag{4.3}$$

where, to estimate the acceleration of star i, the sum is taken over all the interactions with the other stars j. Writing these as first order equations we obtain

$$\begin{aligned} x_i' &= u_i \\ y_i' &= v_i \\ u_i' &= G \sum_{j \neq i} m_j \frac{(x_j - x_i)}{r_{ij}^{3/2}} \\ v_i' &= G \sum_{j \neq i} m_j \frac{(y_j - y_i)}{r_{ij}^{3/2}}, \end{aligned} \tag{4.4}$$

where x_i, u_i, y_i, v_i are the positions and velocities in the x and y directions of star i respectively.

With 7 stars, and four differential equations per star, this problem comprises 28 equations. As in [17], we assume that the masses $m_i = i$ and that the gravitational constant G equals 1; the initial conditions are found in [17]. We integrate the problem in the time interval [0, 3]. In the function that implements the derivative in R (`pleiade`), we start by separating the input vector `Y` into the coordinates (`x, y`) and velocities (`u, v`) of each star. The distances in the x and y directions are created using R function `outer`. This function will apply `FUN` for each combination of `x` and `y`. It thus creates a matrix with 7 rows and 7 columns, having for `distx` on the position i, j, the value $x_i - x_j$. The matrix containing the values $r_{ij}^{3/2}$, called rij3, is then calculated using distx and disty. Finally we multiply matrix distx or disty with the vector containing the masses of the stars (starMass), and divide by matrix rij3. The result of these calculations are two matrices (fx, fy), with 7 rows and columns. As the distance between a body and itself is equal to 0, this matrix has NaN (Not a Number) on the diagonal. The required summation to obtain u' and v' (Eq. (4.3)) is done using R function colSums; the argument na.rm = TRUE ensures that these sums ignore the NaNs on the diagonal of fx and fy. During the movement of the 7 bodies several quasi-collisions occur, where the squared distance between two bodies are as small as 10^{-3}. When this happens, the accelerations u', v' get very high. Thus, over the entire integration interval, there are periods with slow motion and periods of rapid motion, such that this problem can only be efficiently solved with an integrator that uses adaptive time stepping.

As the problem is non-stiff, we solved it with the code cashkarp. We use the function system.time to print the elapsed time required to obtain the solution.

```
library(deSolve)
pleiade <- function (t, Y, pars) {
    x <- Y[1:7]
    y <- Y[8:14]
    u <- Y[15:21]
    v <- Y[22:28]

    distx <- outer(x, x, FUN = function(x, y) x - y)
    disty <- outer(y, y, FUN = function(x, y) x - y)

    rij3 <- (distx^2 + disty^2)^(3/2)

    fx <- starMass * distx / rij3
    fy <- starMass * disty / rij3

    list(c(dx = u,
           dy = v,
           du = colSums(fx, na.rm = TRUE),
           dv = colSums(fy, na.rm = TRUE)))
 }
starMass <- 1:7
yini<- c(x1= 3, x2= 3, x3=-1, x4=-3,     x5= 2, x6=-2,    x7= 2,
         y1= 3, y2=-3, y3= 2, y4= 0,     y5= 0, y6=-4,    y7= 4,
         u1= 0, u2= 0, u3= 0, u4= 0,     u5= 0, u6=1.75, u7=-1.5,
         v1= 0, v2= 0, v3= 0, v4=-1.25, v5= 1, v6= 0,    v7= 0)
print(system.time(
  out <- ode(func = pleiade, parms = NULL, y = yini,
       method = cashkarp, times = seq(from = 0, to = 3, by = 0.01))
       ))
```

```
 user  system elapsed
 0.17    0.00    0.17
```

To use the Fortran interface in the **deTestSet** package the instructions are

```
library(deTestSet)
print(system.time(
  out2 <- pleiades(method = cashkarp, times = seq(0, 3, 0.01)))
  )
```

```
 user  system elapsed
    0       0       0
```

The execution times show that the numerical solution obtained using the R interface to the Fortran code requires much less computational time than the one required by using the R implementation of the model. More about computing performance in pure R and compiled code can be found in [10]. A package vignette of **deSolve** deals with how to write ODE models in compiled code and solve them with R [28].

In Fig. 4 we show the work precision diagrams for the Pleiades problem of the **deTestSet** solvers compared with adams, lsoda and daspk codes from the package **deSolve**.

More information about the Pleiades problem using R can be found in [9].

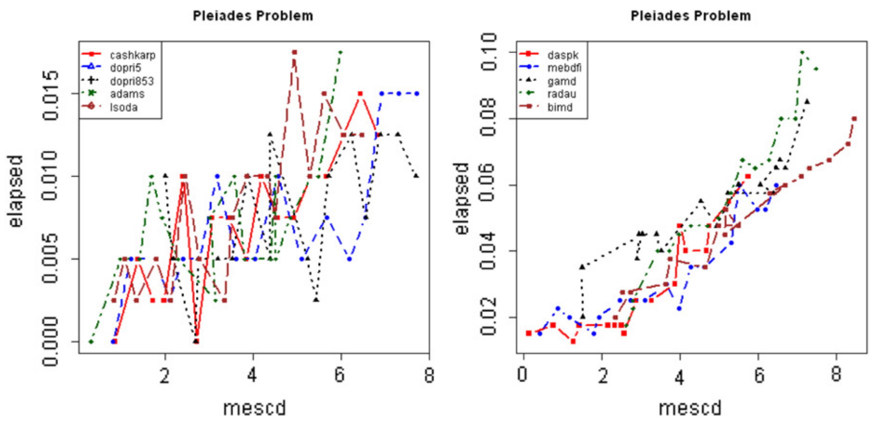

Fig. 4. Work precision diagrams for the Pleiades problem. Explicit solvers (on the left)—Implicit solvers (on the right).

Table 1
Exit flag for the explicit solvers.

Problem	rtol	atol	`cashkarp` (eig)	`cashkarp` (err)	`dopri5`	`dopri853`
Vdpol	$10^{-4-j(5/8)}$	$10^{-4-j(5/8)}$	-4 ($j = 0, \dots, 16$)	-5 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)
Ring	$10^{-4-j(5/8)}$	$10^{-4-j(5/8)}$	-4 ($j = 0, \dots, 16$)	-5 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)
E5	$10^{-4-j(5/8)}$	10^{-24}	-4 ($j = 0, \dots, 16$)	-5 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)
Rober	$10^{-4-j(5/8)}$	$10^{-4-j(5/8)}$	-4 ($j = 0, 1, 3, \dots, 16$)	-2 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-2 ($j = 5$)
			-3 ($j = 2$)	-3 ($j = 1, 2, 3$)		-3 ($j = 0, 1$)
				-5 ($j = 4, \dots, 15$)		-4 ($j = 2, 3, 4, 6, \dots, 16$)
Beam	$10^{-4-j(1/4)}$	$10^{-4-j(1/4)}$	-4 ($j = 0, \dots, 16$)	1 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)
EMEP	$10^{-4-j(3/8)}$	1	-4 ($j = 0, \dots, 16$)	-5 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)
Orego	$10^{-4-j(5/8)}$	$10^{-4-j(5/8)}$	-4 ($j = 0, \dots, 16$)	-5 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)
Pollution	$10^{-4-j(5/8)}$	$10^{-4-j(5/8)}$	-4 ($j = 0, \dots, 16$)	-5 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)
HIRES	$10^{-4-j(5/8)}$	$10^{-4-j(5/8)}$	-4 ($j = 0, \dots, 16$)	-5 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)	-4 ($j = 0, \dots, 16$)

4.1.3. *Stiffness detection*

The Pleiades problem is the only non stiff problem in the IVPTESTSET which can be efficiently solved with the explicit Runge–Kutta solvers. When trying to solve stiff problems, the explicit Runge–Kutta codes usually stop upon detecting stiffness.

In Table 1 we report the exit flag of the codes when we try to solve one of the stiff problems using a fixed set of relative and absolute tolerances. In this table, `cashkarp(eig)` is the code `cashkarp` with only the stiffness detection based on the check of eigenvalues activated, in `cashkarp(err)` the stiffness detection based on error check is used. The meaning of the error flag is as follows:

1: successful stop;
-2: maximum number of steps reached;
-3: stepsize too small;
-4: the problem seems to become stiff using the standard check of eigenvalues:
-5: the problem seems to become stiff using the stiffness detection algorithm based on two error estimators.

Reading Table 1 we see that the behavior of `cashkarp(eig)`, `dopri5` and `dopri853` using the same stiffness detection based on the standard check of eigenvalues is the same. Only for the Robertson problem the codes `cashkarp(eig)` and `dopri853` do not detect stiffness for some input tolerances but exit because the stepsize is either too small or the maximum number of allowed steps has been reached. The stiffness detection based on the error check (implemented only in `cashkarp`) does not detect stiffness for the Beam problem and for the Robertson problem has a similar behavior of the stiffness detection based on the eigenvalues. We can conclude from this that all the stiffness detection algorithms are reliable on these test problems. After stiffness has been detected we can switch to an implicit solver.

4.1.4. *Ring modulator*

The ring modulator problem is a stiff system of 15 ODEs that originates from electrical circuit analysis. The work precision diagram in Fig. 5 (on the left) has been computed using $atol = rtol = 10^{-4-j(5/8)}$, for $j = 0, \dots, 16$. The reference solution has been computed using *bimd* with $atol = rtol = 10^{-14}$, using `times <- seq(0, 0.001, by = 5e-06)`. It shows that the most efficient codes for this problems are `mebdfi` and `bimd`.

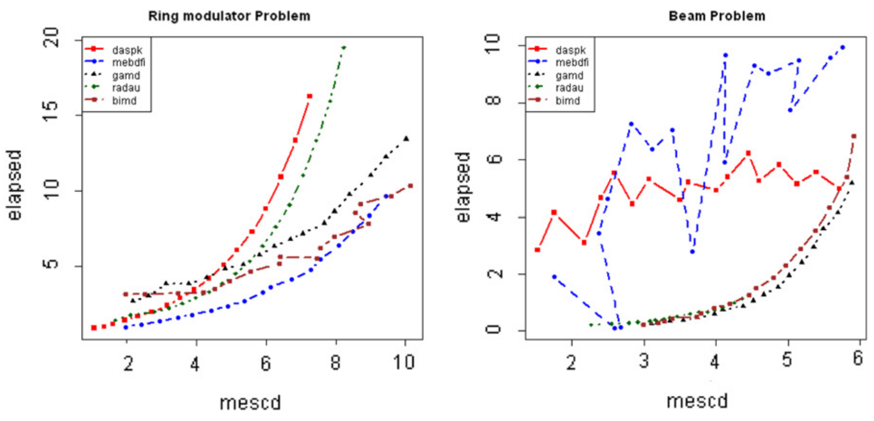

Fig. 5. Work precision diagrams for the ring modulator problem (on the left) and the beam problem (on the right).

4.1.5. Beam problem

The beam problem originates from mechanics and describes the motion of an elastic beam. It is a stiff ODE consisting of 80 differential equations. The work precision diagram in Fig. 5 (on the right) has been computed using $atol = rtol = 10^{-4-j(1/4)}$, for $j = 0, \ldots, 16$. The reference solution has been computed using *bimd* with $atol = rtol = 10^{-10}$, using `times <- seq(0, 5, by = 0.01)`. It shows that the most efficient codes for this problems are `gamd` and `bimd`. Note that for the beam problem the code `mebdfi` works well if the maximum order is limited to 4 (see [16, p. 304]) because in this case the code is based on *A*-stable formulas.

4.2. Numerical experiments: DAEs and IDEs

In this section we present work precision diagrams of selected DAE test problems using the R solvers in the **deSolve** and **deTestSet**. The work precision diagrams are computed using as tolerances $rtol = atol = 10^{-4-j(3/8)}$ for $j = 0, \ldots, 16$. The reference solution has been computed using the solver `bimd` with $rtol = atol = 10^{-12}$. The first problem is the Transistor Amplifier, which is an index 1 DAE of dimension 8 that originates from electrical circuit analysis. The results are reported in Fig. 6 on the left. The figure shows that on this problem the most efficient codes are `daspk` and `mebdfi`. The situation is similar for the Fekete problem, an index two DAE from mechanics (Fig. 6 on the right). For this problem, the codes `mebdfi`, `gamd` and daspk are the most efficient.

The next two problems are both index three DAEs. For these problems the reference solution has been computed using the solver `bimd` with $rtol = atol = 10^{-14}$. The caraxis problem is of size 10, and is an example of a multibody system. For this problem the most efficient codes using as tolerances $rtol = atol = 10^{-4-j(3/8)}$ for $j = 0, \ldots, 16$ and `times <- seq(0,3,by=3/500)` (see Fig. 7) are `mebdfi` and `radau`, we observe that even though `daspk` is not designed to solve index three DAEs it is able to compute the solution.[2] The Andrews squeezing mechanism is of size 27 and describes the motion of 7 rigid bodies connected by joints without friction. In Fig. 7 we report the results using as tolerances $rtol = atol = 10^{-4-j(3/8)}$ for $j = 0, \ldots, 16$ and `times <- seq(0,0.03,by=0.03/500)`. The behavior is similar to the caraxis problem, the only difference being that `daspk` is not able to obtain a solution.

The last example is an IDE problem of index 2, called Wheelset, which is of size 17 and is related to the simulation of problems in contact mechanics. Since it is formulated as an IDE, only `daspk` and `mebdfi` could be used for the solution. The behavior of the two codes is very similar with `daspk` the most efficient (see Fig. 8, the work precision diagrams has been computed using $rtol = atol = 10^{-4-j(3/8)}$ for $j = 0, \ldots, 16$, `times <- seq(0,10,10/500)`), and the reference solution has been computed using the solver `mebdfi` with $rtol = atol = 10^{-14}$.

5. Final remarks

In order to make the IVPTESTSET available to people who are not well acquainted with Fortran, a Matlab interface to the test set was written in 2006. In addition, some of the test problems have also been inserted in the OCTAVE `odepkg` package (http://octave.sourceforge.net/odepkg/).

With the new R package **deTestSet**, users of the open source software R now can also benefit from this facility.

In addition to the interface to IVPTESTSET, the new package provides some efficient integration codes that add to the repertoire of methods already available in the R package **deSolve**.

Potential users of the R package **deTestSet** are scientists and engineers who either need to solve differential equations or need to simulate, on a computer, scientific problems based on differential equations. Teachers may also find both the code

[2] This is because in the R implementation of DASPK2, the same scaling strategy as in `radau` has been implemented—see [9].

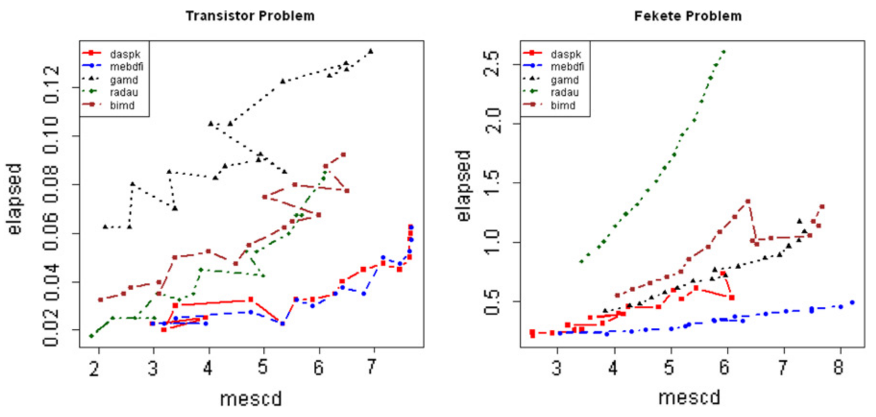

Fig. 6. Work precision diagrams for the Transistor problem (on the left) and the Fekete problem (on the right), both are index 1 DAEs.

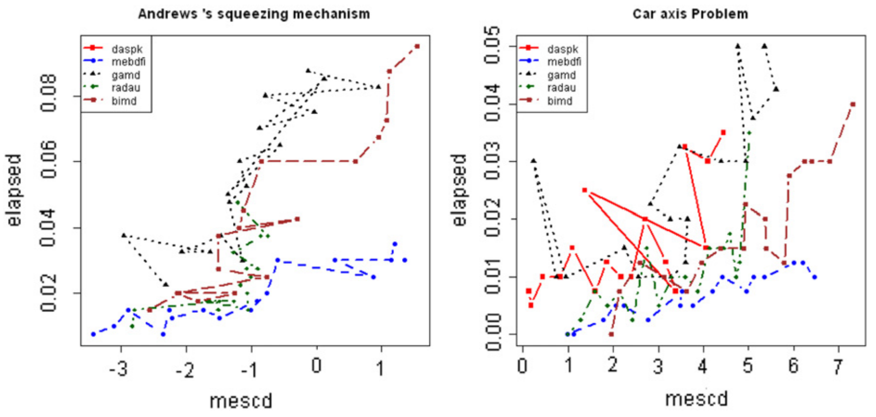

Fig. 7. Work precision diagrams for the Andrews problem (on the left) and the caraxis problem (on the right), both are index 2 DAEs.

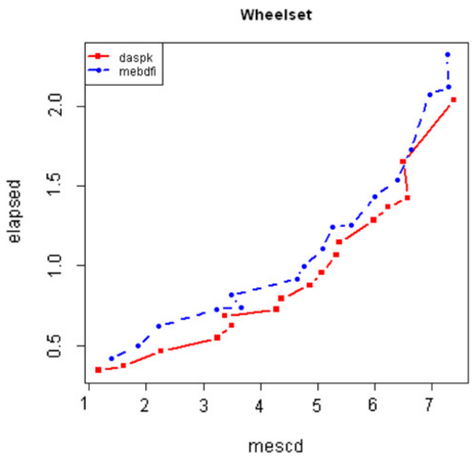

Fig. 8. Work precision diagrams for the Wheelset problem, IDE of index 2.

and problem data bases useful when organizing courses concerning the numerical simulation on newly emerging fields of experimental mathematics.

Acknowledgments

The work was developed within the project "Numerical methods and software for differential equations". The present paper relies very heavily on the pioneering work of P. van der Houwen (CWI, Amsterdam), who initiated the IVPTESTSET, originally sponsored by STW grants CWI.4533 and CWI.2703.

References

[1] W.M. Lioen, J.J.B. de Swart, W.A. van der Veen, Test set for IVP solvers, Tech. Rep. NM-R9615, CWI, Amsterdam, 1996.
[2] F. Mazzia, C. Magherini, Test set for initial value problem solvers, release 2.4, Report 4/2008, Department of Mathematics, University of Bari, Italy, 2008. URL: http://pitagora.dm.uniba.it/~testset.
[3] T. Mathworks, Matlab release 2011b, URL: http://www.mathworks.com/.
[4] I. Wolfram Research, Mathematica Edition, Version 8.0, Wolfram, Research Inc., 2010.
[5] M.B. Monagan, K.O. Geddes, K.M. Heal, G. Labahn, S.M. Vorkoetter, J. McCarron, P. DeMarco, Maple10 Programming Guide. Maplesoft, Waterloo ON, Canada, 2005.
[6] Scilab Consortium: Scilab: The free software for numerical computation, Scilab Consortium, Digiteo, Paris, France, 2011. URL: http://www.scilab.org.
[7] J.W. Eaton, GNU Octave Manual. Network Theory Limited, 2002. ISBN: 0-9541617-2-6. URL: http://www.octave.org.
[8] R Development Core Team: R: a language and environment for statistical computing, R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN: 3-900051-07-0. URL: http://www.R-project.org/.
[9] K. Soetaert, J.R. Cash, F. Mazzia, Solving Differential Equations in R, Springer, ISBN: 978-3-642-28069-6, 2012.
[10] K. Soetaert, T. Petzoldt, R.W. Setzer, Solving differential equations in R: package deSolve, Journal of Statistical Software 33 (9) (2010) 1–25. URL: http://www.jstatsoft.org/v33/i09.
[11] K. Soetaert, RootSolve: nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations, 2009. R package version1.6. URL: http://CRAN.R-project.org/package=rootSolve.
[12] K. Soetaert, F. Meysman, Reactive transport in aquatic ecosystems: rapid model prototyping in the open source software R, Environmental Modelling & Software 32 (2012) 49–60. http://dx.doi.org/10.1016/j.envsoft.2011.08.011.
[13] K. Soetaert, J.R. Cash, F. Mazzia, BvpSolve: solvers for boundary value problems of ordinary differential equations, 2010. R package version 1.2 URL: http://CRAN.R-project.org/package=bvpSolve.
[14] S. Theußl, A. Zeileis, Collaborative software development using R-forge, The R Journal 1 (1) (2009) 9–14. URL: http://journal.r-project.org/2009-1/RJournal_2009-1_Theussl+Zeileis.pdf.
[15] J.R. Cash, S. Considine, An MEBDF code for stiff initial value problems, ACM Transactions on Mathematical Software 18 (2) (1992) 142–158.
[16] E. Hairer, G. Wanner, Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, Springer-Verlag, Heidelberg, 1996.
[17] E. Hairer, S.P. Norsett, G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, second revised ed., Springer-Verlag, Heidelberg, 2009.
[18] J.R. Cash, A.H. Karp, A variable order Runge–Kutta method for initial value problems with rapidly varying right-hand sides, ACM Transactions on Mathematical Software 16 (1990) 201–222.
[19] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in FORTRAN 77. The Art of Scientific Computing, second ed., Cambridge University Press, 1992.
[20] L.R. Petzold, Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations, SIAM Journal on Scientific and Statistical Computing 4 (1983) 136–148.
[21] A.C. Hindmarsh, ODEPACK, a systematized collection of ODE solvers, in: R. Stepleman (Ed.), Scientific Computing, in: IMACS Transactions on Scientific Computation, vol. 1, IMACS, North-Holland, Amsterdam, 1983, pp. 55–64.
[22] P.N. Brown, G.D. Byrne, A.C. Hindmarsh, VODE, a variable-coefficient ODE solver, SIAM Journal on Scientific and Statistical Computing 10 (1989) 1038–1051.
[23] K.E. Brenan, S.L. Campbell, L.R. Petzold, Numerical solution of initial-value problems in differential–algebraic equations, SIAM: Classics in Applied Mathematics (1996).
[24] F. Iavernaro, F. Mazzia, Solving ordinary differential equations by generalized Adams methods: properties and implementation techniques, Applied Numerical Mathematics 28 (2–4) (1998) 107–126. URL: http://dx.doi.org/10.1016/S0168-9274(98)00039-7. Eighth Conference on the Numerical Treatment of Differential Equations (Alexisbad, 1997).
[25] L. Brugnano, C. Magherini, The BiM code for the numerical solution of ODEs, in: Proceedings of the 10th International Congress on Computational and Applied Mathematics (ICCAM-2002), vol. 164/165, 2004, pp. 145–158. URL: http://dx.doi.org/10.1016/j.cam.2003.09.004.
[26] L. Brugnano, C. Magherini, F. Mugnai, Blended implicit methods for the numerical solution of DAE problems, Journal of Computational and Applied Mathematics 189 (1–2) (2006) 34–50. URL: http://dx.doi.org/10.1016/j.cam.2005.05.005.
[27] L. Shampine, K. Hiebert, Detecting stiffness with the Fehlberg (4, 5) formulas, Computers & Mathematics with Applications 3 (1) (1977) 41–46. URL: http://www.sciencedirect.com/science/article/pii/0898122177901122.
[28] K. Soetaert, T. Petzoldt, R.W. Setzer, R-package deSolve, writing code in compiled languages, Package vignette, 2009. URL: http://CRAN.R-project.org/package=deSolve.