Opuscula Math. 34, no. 2 (2014), 387–403
http://dx.doi.org/10.7494/OpMath.2014.34.2.387 Opuscula Mathematica

Dedicated to the Memory of Professor Zdzisław Kamont

SOLVING BOUNDARY VALUE PROBLEMS IN THE OPEN SOURCE SOFTWARE R: PACKAGE bvpSolve

Francesca Mazzia, Jeff R. Cash, and Karline Soetaert

Communicated by Zdzisław Jackiewicz

Abstract. The R package **bvpSolve** for the numerical solution of Boundary Value Problems (BVPs) is presented. This package is free software which is distributed under the GNU General Public License, as part of the R open source software project. It includes some well known codes to solve boundary value problems of ordinary differential equations (ODEs) and differential algebraic equations (DAEs). In addition to the packages already available for solving initial value problems, the new package now allows non expert users to efficiently solve boundary value problems in the problem solving environment R.

Keywords: ordinary differential equations, boundary value problems, singular perturbation problems, test problems, R, Fortran.

Mathematics Subject Classification: 65L04, 65L05, 65L80.

1. INTRODUCTION

In this paper we present the R package **bvpSolve** for the numerical solution of Boundary Value Problems (BVPs). The package includes solvers for Boundary Value Problems (BVPs) of Ordinary Differential Equations and Differential Algebraic Equations.

A generic two-point boundary value problem for ordinary differential equations is a system of ordinary differential equations whose solution is subject to conditions posed at two distinct points in the range of integration. It is possible to have higher order equations with boundary conditions posed at more than two distinct points.

The generic problem is:

$$y''(x) = f(x, y, y'), \quad a \le x \le b, \tag{1.1}$$

where $y \in \mathbb{R}^m$, $f\colon \mathbb{R} \times \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m$, with boundary conditions,

$$g(y(a), y(b)) = 0.$$

This general form for the boundary conditions allows both separated and non-separated conditions to be specified.

Two point BVPs originate in a number of areas including fluid flow, shock waves, epidemiology and geophysical models, see [2] and [32] for more examples.

Apart from the engineers and scientists who 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 [22], Mathematica [39] or Maple [28], or open-source software such as Scilab [30] or Octave [16].

One of the emerging PSEs whose use is expanding very rapidly, especially in universities and academia, is the open source software R [29]. Although still mainly known as software for visualisation and statistics, R has recently been extended to also provide powerful methods for solving differential equations [32] by a number of extension packages. The R packages **deSolve** [37] and **deTestSet** [34] provide solution methods and test problems for initial value problems of ODEs, DDEs, PDEs and DAEs. The R packages **rootSolve** [31] and **ReacTran** [35] provide more solution methods for PDEs.

Most of the BVP solvers are written in Fortran and based on advanced numerical techniques for differential equations. The new package **bvpSolve** has implemented these well known Fortran codes for the solution of BVPs and provides an interface to these codes.

The available Fortran codes are `twpbvp.f` [15], `twpbvpl.f` [6] and `acdc.f` [14], their variant based on conditioning called `twpbvpc.f` [11] and `twpbvplc.f` [12] and `acdcc.f`, and the collocation codes `colsys.f` [1], `colnew.f` [5], `colmod.f` [14] and `coldae` [4].

We recall that there are other important codes that are not yet included in the R package **bvpSolve**. The are the Fortran codes such as `mirkdc.f` [17] and `BVP_M-2.f90`, based on MIRK methods, and the Matlab code TOM [24–26].

We note that the Matlab code `bvptwp` [10] is based on the same Fortran codes as implemented in **bvpSolve**, but the Matlab version of these codes consists of an efficient translation of the Fortran codes, whereas the R version provides an interface to the Fortran codes.

This paper is structured as follows. First, in Section 2, we define the classes of problems that we can solve using new R package. Section 3 introduces the integration routines available in the package **bvpSolve**, while Section 4 gives some information about the conditioning parameters used by `bvptwp`. Section 5 gives some examples of implementation in R with numerical benchmarks of computational performance. Finally some concluding remarks are given in Section 6. The package **bvpSolve** is available from the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=bvpSolve`. New versions of the package, still under development, are available at `http://r-forge.r-project.org/` [38].

2. CLASSES OF PROBLEMS

The test problems that can be solved using the R package **bvpSolve** can be categorized into the following classes:

- systems of first order and higher order two point Boundary Value Problems (BVPs),
- singularly perturbed Boundary Value Problems (SPBVPs) and
- Differential Algebraic Boundary Value Problems (DABVP).

We call a problem a multipoint mixed order BVP if it has the form

$$\begin{array}{l} y_i^{r_i} = f_i(x, z(y(x))), \quad i = 1, d,\ a \le x \le b, \\ y, f \in \mathbb{R}^d,\ z(y(x)) = (y(x)_1, y(x)'_1, \ldots, y(x)_1^{r_1}, \ldots, y(x)_d, y(x)'_d, \ldots, y(x)_d^{r_d}), \\ g_j(z(y(x_j))) \quad \text{given}, \quad a \le x_j \le b, j = 1, \ldots, \sum_{i=1}^{d} r_i, \end{array} \tag{2.1}$$

where r_i is the order of the i-th differential equation.

A multipoint mixed order SPBVP is of the form:

$$\begin{array}{l} y_i^{r_i} = f_i(x, z(y(x)), \epsilon), \quad i = 1, d,\ a \le x \le b, \\ y, f \in \mathbb{R}^d, z(y(x)) = (y(x)_1, y(x)'_1, \ldots, y(x)_1^{r_1}, \ldots, y(x)_d, y(x)'_d, \ldots, y(x)_d^{r_d}), \\ g_j(z(y(x_j))) \quad \text{given}, \quad a \le x_j \le b, j = 1, \ldots, \sum_{i=1}^{d} r_i, \end{array} \tag{2.2}$$

where ϵ is a small parameter $\epsilon > 0$.

A problem is called a multipoint mixed order DABVP if it is of the form:

$$\begin{array}{l} y_i^{r_i} = f_i(x, z(y(x)), u(x)), \quad i = 1, d,\ a \le x \le b, \\ 0 = f_i(x, z(y(x)), u(x)), \quad i = d+1, l,\ a \le x \le b, \\ y, f \in \mathbb{R}^d, u \in \mathbb{R}^l, z(y(x)) = (y(x)_1, y(x)'_1, \ldots, y(x)_1^{r_1}, \ldots, y(x)_d, y(x)'_d, \ldots, y(x)_d^{r_d}), \\ g_j(z(y(x_j))) \quad \text{given}, \quad a \le x_j \le b, j = 1, \ldots, \sum_{i=1}^{d} r_i. \end{array} \tag{2.3}$$

Problems described by (2.1), (2.2) and (2.3) include the class of two point first order problems, where the boundary conditions are given only at a and b.

Only a few available codes can solve mixed order BVPs directly, i.e. without reducing the problem to first order form or can handle multipoint boundary conditions.

However, such problems can easily be transformed in their first order form with two point separated boundary conditions. This form can be handled by all the available codes:

$$\begin{array}{l} y(x)' = f(x, y(x)), \quad a \le x \le b, \\ y, f \in \mathbb{R}^d, \\ B_a y(a) + B_b y(b) \quad \text{given}, \quad B_a, B_b \in R^{d \times d}. \end{array} \tag{2.4}$$

3. THE INTEGRATION ROUTINES

The R package **bvpSolve** [33] is closely related to the packages **deSolve** [37] and **deTestSet** [34]. 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. Thus, whereas the underlying Fortran codes have quite different calling interfaces, the R interface is the same for the different classes of solvers. Hence, once a problem is implemented in R, it is simple to invoke each of the solvers, in order to select the most efficient one.

3.1. FINITE DIFFERENCE SOLVERS

One finite difference solver, available in **bvpSolve** provides an interface to the Fortran codes `twpbvpc.f` [11], `twpbvplc.f` [12], `twpbvp.f` [15], and `twpbvpl.f` [6].

The code `twpbvpc.f` uses a deferred correction scheme based on Mono-Implicit Runge-Kutta methods (MIRK) [18]; the other code uses a deferred correction scheme based on Lobatto formulas. They are an improved implementation of the codes `twpbvp.f` and `twpbvpl.f` which monitors the conditioning.

The R function that calls the interfaces to these codes is `bvptwp`. On changing some input parameters, the various codes are invoked.

The simplest calling sequence for solving a BVP in R using a finite difference solver is:

```
bvptwp(yini, x, yend, func, parms, ...)
```

where `x` holds the sequence of the independent varaiable at which output is wanted, `yini` holds the initial conditions, `yend` holds the final conditions, `func` is the R function that describes the differential equations, and `parms` contains the parameter values (or is `NULL`).

The default integration solver for the function `bvptwp` is `twpbvp`. To solve a problem using the other methods we should specify some input parameters. For example to use the code `twpbvpl.f` we must set `lobatto = TRUE`, to use the version of the codes that use the mesh selection based on conditioning we must set `cond = TRUE`.

If we type `?bvptwp` 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.

The full set of arguments to `bvptwp` is:

```
function (yini = NULL, x, func, yend = NULL, parms = NULL,
    order = NULL, ynames = NULL, xguess = NULL, yguess = NULL,
    jacfunc = NULL, bound = NULL, jacbound = NULL, leftbc = NULL,
    posbound = NULL, islin = FALSE, nmax = 1000, ncomp = NULL,
    atol = 1e-08, cond = FALSE, lobatto = FALSE, allpoints = TRUE,
    dllname = NULL, initfunc = dllname, rpar = NULL, ipar = NULL,
    nout = 0, forcings = NULL, initforc = NULL, fcontrol = NULL,
    verbose = FALSE, epsini = NULL, eps = epsini,
    ...)
NULL
```

Many additional inputs can be provided, e.g. the error tolerances (defaults `atol = 1e-8`), initial guesses of x and y (`xguess, yguess`), etc... Note that a more complex interface, as required in all underlying codes can also be provided, by the arguments

`bound` (function that defines the boundary conditions), `jacbound` (function with the Jacobian of the boundary conditions), `jacfunc` (function that calculates the Jacobian of the problem definition). Also note that, in accordance to the IVP integration methods in **deSolve**, the BVP problem can also be implemented in compiled code (argument `dllname`) (see below).

3.2. COLLOCATION SOLVERS

The collocation solvers available in the package **bvpSolve** are `colsys` [1], `colnew` [5] and `coldae` [4].

The codes `colsys.f/colnew.f` are based on the approximation of the solution of the differential equation by a piecewise polynomial and it uses collocation at Gauss points to define this polynomial uniquely. The error control in `colsys.f/colnew.f` is based on an estimate of the discretization error and an estimate of the global error of a continuous solution approximation. The code `coldae` is a generalization of `colnew` for the solution of Differential Algebraic Equations (DAEs) [4].

The R function that calls the interface is `bvpcol`. The simplest calling sequence for solving BVPs with these methods is the same as for the finite difference solvers:

```
bvpcol(yini, x, func, yend, parms, ...)
```

This calls the code `colnew`.

To call `colsys` we need to set the parameter `bspline=TRUE`, to call `coldae` we need to give as input a list containing the index of the problem and the number of algebraic equations l in (2.3), in this case the function defining the problem includes the algebraic equations.

The full set of arguments of `bvpcol` is:

```
function (yini = NULL, x, func, yend = NULL, parms = NULL,
    order = NULL, ynames = NULL, xguess = NULL, yguess = NULL,
    jacfunc = NULL, bound = NULL, jacbound = NULL, leftbc = NULL,
    posbound = NULL, islin = FALSE, nmax = 1000, ncomp = NULL,
    atol = 1e-08, colp = NULL, bspline = FALSE, fullOut = TRUE,
    dllname = NULL, initfunc = dllname, rpar = NULL, ipar = NULL,
    nout = 0, forcings = NULL, initforc = NULL, fcontrol = NULL,
    verbose = FALSE, epsini = NULL, eps = epsini, dae = NULL, ...)
NULL
```

3.3. AUTOMATIC CONTINUATION SOLVERS

The automatic continuation solvers available in the package **bvpSolve** are the codes `colmod.f` and `acdcc.f`. The first one is based on the continuation code `colnew.f` and is implemented in the R function `bvpcol`, the second one is based on the finite difference code `twpbvpl.f` and is implemented in the R function `bvptwp`.

To use these methods the problem should depend on a small parameter ϵ. This should be an input parameter and the first parameter in parms.

The simpler calling sequence for solving BVPs with `colmod.f` is

```
parms <- eps bvpcol(yini, x, func, yend, parms, eps = parms, ...)
```

To use `acdcc.f` the calling sequence is

```
parms <- eps bvptwp(yini, x, func, yend, parms, eps = parms, ...)
```

3.4. A SHOOTING SOLVER

The R function `bvpshoot` is an R implementation of the single shooting method. This combines the integration routines from the package **deSolve** [37] with root-finding methods from package **rootSolve** [31]. The simpler calling sequence of `bvpshoot` is the same as `bvpcol` and `bvptwp`

```
library(bvpSolve) bvpshoot(yini, x, func, yend, parms, ...)
```

4. CONDITIONING PARAMETERS

The R function `bvptwp` gives as output information about the conditioning parameters. We recall that the concept of stability, which is usually applied to initial value problems for differential equations, is referred to as conditioning for boundary value problems. Conditioning relates to the effect small changes in equation (2.1), either in the function f, or in the boundary conditions $g(y(a), y(b))$, have on the solution. It is very important to obtain an estimate of the conditioning parameters when computing a solution of a BVP, since a small local error does not necessarily give rise to a small global error.

In order to obtain a more precise feeling for the concept of a stable boundary value problem, we refer the reader to [32] and [3].

In particular, if we consider the following linear boundary value problem:

$$\frac{dy}{dx} = A(x)y(x) + q(x), \quad a \le x \le b, \quad B_a y(a) + B_b y(b) = \beta, \quad \beta \in \mathbb{R}^m, \tag{4.1}$$

and a perturbed equation:

$$\frac{du}{dx} = A(x)u(x) + q(x) + \delta(x), \quad a \le x \le b, \quad B_a u(a) + B_b u(b) = \beta + \delta\beta. \tag{4.2}$$

(here $\delta(x)$ and $\delta\beta$ are small perturbations of the data) the output given by the codes is an approximation of the parameters $\kappa, \kappa_1, \kappa_2, \gamma_1$ and σ defined below:

$$\kappa_1 = \max_{a \le x \le b} \|Y(x)Q^{-1}\|, \qquad \kappa_2 = \sup_x \int_a^b \|G(x,t)\| dt, \tag{4.3}$$

and

$$\kappa = \max_{a \le x \le b} \left(\|Y(x)Q^{-1}\| + \int_a^b \|G(x,t)\| dt \right). \tag{4.4}$$

Here $Y(x)$ is a fundamental solution, $Q = B_a Y(a) + B_b Y(b)$ is non singular and $G(x,t)$ is the Greens' function. These parameters are a bound of the absolute error using the ∞-norm:

$$\max_{a \le x \le b} \|u(x) - y(x)\| \le \kappa_1 \|\delta\beta\| + \kappa_2 \max_{a \le x \le b} \|\delta(x)\|, \tag{4.5}$$

and

$$\max_{a \le x \le b} \|u(x) - y(x)\| \le \kappa \max(\|\delta\beta\|, \max_{a \le x \le b} \|\delta(x)\|), \tag{4.6}$$

Using the 1-norm we obtain the corresponding conditioning parameters called γ_1, γ_2 and γ. Another important parameter is σ, which is called the "stiffness ratio". It is defined for linear problems (4.1) as

$$\sigma = \max_{\delta\beta} \frac{\max_{a \le x \le b} \|u(x) - y(x)\|}{\frac{\int_a^b \|u(x) - y(x)\| dx}{(b-a)}}. \tag{4.7}$$

where $u(x)$ is the solution of the perturbed equation with $\delta(x) \equiv 0$ (see [13,23,27]). If σ is large we are dealing with problems possessing different time scales for which the growth or decay rates of some fundamental solution modes are very rapid compared to others, see [7,8,21,23] for details.

5. THE R IMPLEMENTATION OF A TEST PROBLEM

We now describe one example in detail, so the user can learn how to solve a BVP problem using R. The reader is referred to [32] for more examples. We choose the first test problem called `bvpT1` from the testset in [9] It is defined by the following second order differential equation:

$$\epsilon y'' - y = 0, \qquad y(0) = 1,\ y(1) = 0, \tag{5.1}$$

where ϵ is a parameter, the solution has a boundary layer of width $O(\sqrt{\epsilon})$ at $x = 0$.

The FORTRAN codes used in `bvptwp` allow the solution of a first order system only. Although in the R implementation, a higher order problem is automatically converted to a system of first order problems, this is not very efficient in general.

Hence we have implemented for this test both the first order and the second order form of the problem. We convert (5.1) into a first order system of ODEs by adding an extra variable, representing the first order derivative:

$$\begin{aligned} y_1' &= y_2, \\ y_2' &= y_1/\epsilon. \end{aligned} \tag{5.2}$$

Singularly perturbed problems are obtained for small ϵ. We ran the model for ϵ = 1e-4, with relative and absolute tolerances `atol=1e-4`. We can implement this problem in R by using the simplest calling sequence in the following way:

```
yini  <- c(1, NA)
yend  <- c(NA, 0)
feval <- function(x, y, eps)
    return(list(c(y[2],
                  y[1]/eps)))
```

Here `NA` denotes that the boundary condition is not available. The values of the independent variable, where we want to inspect output are:

```
x <- seq(from = 0, to = 1, by = 0.1)
```

After specifying the parameter ϵ, a solution is obtained by invoking the solver:

```
eps <- 1e-4
sol  <- bvptwp(yini = yini, x = x, func = feval, yend = yend,
           parms = eps, atol=1e-4)
```

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 succesfull, the number of function evaluations, the conditioning parameters and so on.

```
diagnostics(sol)
```

```
--------------------
solved with  bvptwp
--------------------
  Integration was successful.

  1 The return code                                 : 0
  2 The number of function evaluations              : 3377
  3 The number of jacobian evaluations              : 404
  4 The number of boundary evaluations              : 42
  5 The number of boundary jacobian evaluations     : 16
  6 The number of steps                             : 33
  7 The number of mesh resets                       : 1
  8 The maximal number of mesh points               : 1000
  9 The actual number of mesh points                : 30
 10 The size of the real work array                 : 56108
 11 The size of the integer work array              : 6006

--------------------
conditioning pars
--------------------
```

```
 1 kappa1  : 100
 2 gamma1  : 1.488746
 3 sigma   : 74.32533
 4 kappa   : 101.01
 5 kappa2  : 1.01
```

To plot the solution we write:

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

This simple statement plots all 2 dependent variables constituting the bvpT1 problem at once, using a line width twice the size of the default (`lwd`).

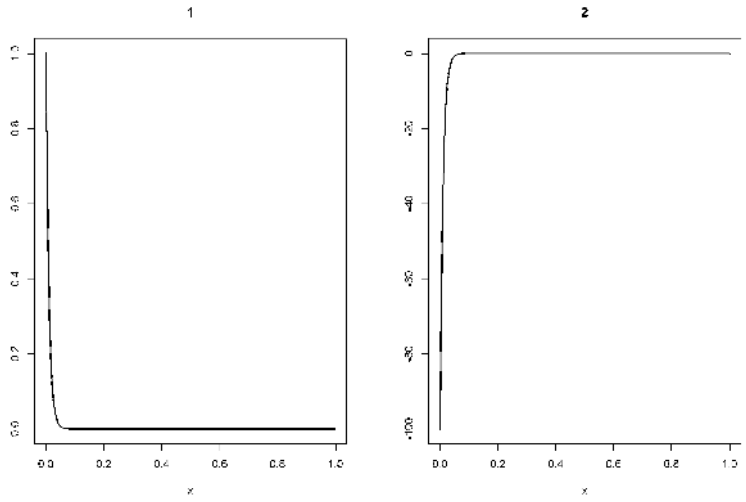

Fig. 1. Numerical solution of the bvpT1 problem. See text for the R code.

If we would like to use instead the code `twpbvplc` we add some input parameters and the input sequence will be:

```
eps <- 1e-4
sol2  <- bvptwp(yini = yini, x = x, func = feval, yend = yend,
           parms = eps, atol = 1e-4, lobatto = TRUE, cond = TRUE)
```

We can also use the default collocation method:

```
sol3  <- bvpcol(yini = yini, x = x, func = feval, yend = yend,
           parms = eps, atol = 1e-4)
diagnostics(sol3)
```

```
--------------------
solved with  bvpcol
--------------------
```

```
Integration was successful.

 1 The return code                                    : 1
 2 The number of function evaluations                 : 1661
 3 The number of jacobian evaluations                 : 280
 4 The number of boundary evaluations                 : 40
 5 The number of boundary jacobian evaluations        : 14
 6 The number of steps                                : 20
 7 The actual number of mesh points                   : 18
 8 The number of collocation points per subinterval   : 4
 9 The number of equations                            : 2
10 The number of components (variables)               : 2
```

Note that we have not used the default value of the tolerance because we want to stress the importance of this parameter to the users.

Using the parameter $\epsilon = 10^{-4}$ the code `bvpshoot` is not able to give a solution. Indeed, for singularly perturbed problems, we should use preferentially `bvptwp` and `bvpcol`, preferably with automatic continuation toggled on (we will calls the codes `acdcc` and `colmod`). To use automatic continuation, the users need only to add to the calling sequence, the parameter `eps`

```
eps <- 1e-8
sol5  <- bvptwp(yini = yini, x = x, func = feval, yend = yend,
           parms = eps, eps = eps, atol = 1e-4)
sol6  <- bvpcol(yini = yini, x = x, func = feval, yend = yend,
           parms = eps, eps = eps, atol = 1e-4)
```

We observe that, for some difficult singular perturbation problems, the automatic continuation codes are able to give a solution whereas the other codes fail.

5.1. THE BOUNDARY CONDITIONS

An important input parameter for all the underlying BVP codes is the function `bound` and `jacbound` that allow to give as input more complicated boundary conditions, including multipoint boundary conditions.

In order for the R functions to also solve these more complex problems, it is also possible to define the boundary conditions using a similar interface, instead of via arguments `yini` and `yend`.

The function `bound` should have the following structure:

```
bound <- function(i, y, parms, ...) { }
```

The output is the j-th boundary condition as defined in (2.1). It is also possible, but not mandatory to use a function that calculates the jacobian with respect to the boundaries. If it is not given, then the R functions will estimate a numerical approximation.

For the problem `bvpT1` the `bound` function is

```
bound <- function(i, y, eps) {
               if (i == 1) return(y[1]-1)
               if (i == 2) return(y[1])
 }
```

When `bound` is used, it becomes necessary to specify the number of equations (`ncomp`), and the number of left boundary conditions (`leftbc`), and the calling sequence becomes:

```
eps <- 1e-4
solbound  <- bvptwp( x = x, func = feval, bound=bound, ncomp=2,
           leftbc=1, parms=eps,  atol=1e-4)
```

Observe that now we do not need to give as input `yini` and `yend`.

5.2. THE ANALYTIC JACOBIANS

When using the simplified (standard) interface, the R functions `bvptwp` and `bvpcol` computate a numerical Jacobian of both the functions f and g in (2.1), (2.2), or (2.3).

For many problems, it may be much more efficient to explicitly provide the analytic Jacobian of both f and g. This as possible giving in input the function `jacfunc` and `jacbound`.

The function `jacfunc` has the following calling sequence:

```
jacfunc = function(x,y,parms, ...){ }
```

and gives as output the matrix with the Jacobian. The function `jacbound` has the following calling sequence

```
jacbound = function(i, y, parms, ...){ }
```

and gives as output the vector with $\partial g_i(y)/\partial y_j$, $j = 1, d$.

For the problem `bvpT1` we have the following functions:

```
jacfunc = function(x, y, eps){
        dfy <- matrix(nrow = 2, ncol = 2, byrow = TRUE,
                  data = c(0,     1,
                           1/eps, 0))
   return(dfy)
 }
jacbound <- function(i, y, eps) {
                  if (i == 1) return(c(1, 0))
                  if (i == 2) return(c(1, 0))
 }
```

The calling sequence that uses both analytic Jacobians is

```
eps <- 1e-4
solajac  <- bvpcol( x = x, func = feval, bound = bound, ncomp = 2,
           leftbc = 1, jacfunc = jacfunc, jacbound=jacbound,
           parms = eps, atol = 1e-4)
```

5.3. MIXED ERROR SIGNIFICANT DIGITS

All the codes in **bvpSolve** 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-8`.

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

$$mescd := -\log_{10}(\max(|absolute\ error/(atol/rtol + |ytrue|)|)), \tag{5.3}$$

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 the exact solution or a more accurate solution computed using the same solver 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 implementations.

5.4. PERFORMANCE AND COMPARISON

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. We took the average of the elapsed CPU times of 4 runs. The format of these diagrams is as in [19, 20, pp. 166–167, 324–325]. As an example we report in Figure 2 (on the left) the work precision diagrams for the methods `twpbvpc`, `twpbvplc`, `colnew`, `colmod`, `acdcc`. running the bvpT1 problem written using the analytical Jacobians with $\epsilon = 1e-5$. The range of tolerances used is, for all codes, $rtol = 10^{-4-j(3/8)}$ with $j = 0, \ldots, 16$, all the other parameters are the default.

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 1.40GHz, 2,80 GB of RAM) and MICROSOFT WINDOWS XP.

Since it is possible to use the second order formulation, we report in Figure 2 (on the right) the work precision diagram implementing the problem using the higher order formulation, which is automatically converted to first order when using `bvptwp`. We observe that the second order formulation reduces the time for the codes `colsys/colnew/colmod` but the execution time for the other codes is higher.

For the bvpT1 problem for $\epsilon = 1e-5$ `twpbvpc` is the most efficient code (Fig. 2), while for the second order implementation the codes `colsys/colnew` require the least computational effort to compute a solution with a similar number of *mescd* (Fig. 2, on the right).

One very useful facility of using R for solving differential equations, is that it is possible to write the problem using a compiled language such us Fortran, C or C++.

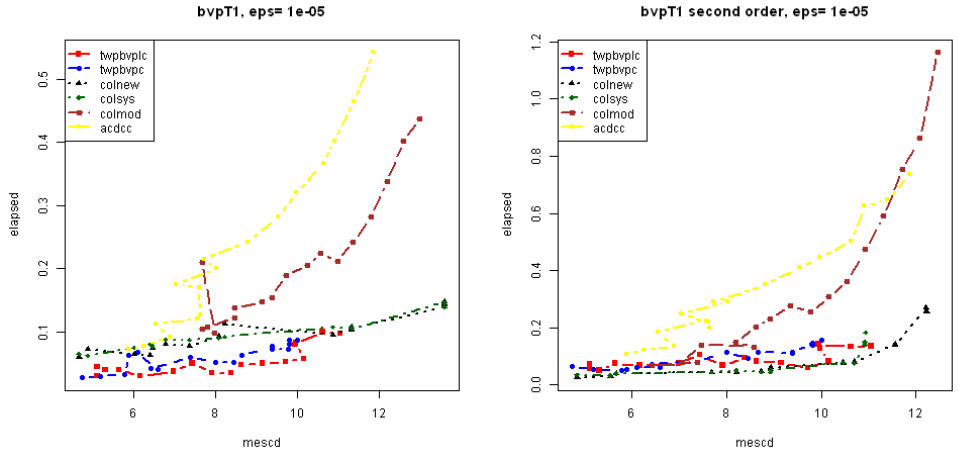

Fig. 2. Work precision diagrams for the bvpT1.R problem, $\epsilon = 1e-5$, first order system (on the left) second order implementation (on the right)

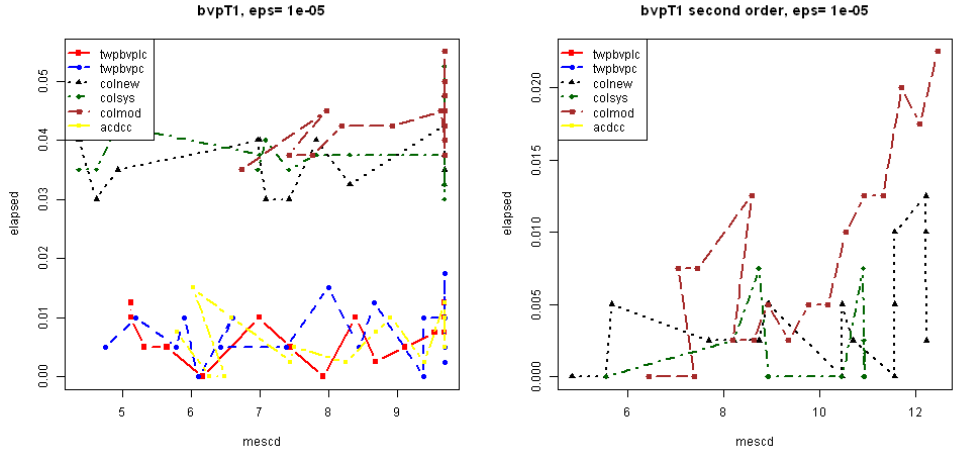

Fig. 3. Work precision diagrams for the bvpT1.f problem, $\epsilon = 1e-5$, first order system (on the left) second order implementation (on the right)

For problems that require a high number of function evaluations, implementing it in compiled code significantly reduces the execution time. See [36] for a description on how to write an IVP problem using a compiled language. How to implement a BVP problem in compiled code is presented in the **bvpSolve** package vignette. Typing `vignette("bvpSolve")` will open this vignette.

We now run the code using the test problem written in Fortran. We observe that, the behaviour of the solvers changes using the two different implementations see Figure 3 (on the left) for the first order implementation and Figure 3 (on the right) for the second order one. In particular the execution time of the collocation codes is considerable higher compared to the finite difference codes. The collocation codes, in fact, use a number of function evaluations that is considerably smaller with respect to the one used by the finite difference codes, this explains the difference in time using the compiled version of the problem.

6. FINAL REMARKS

The package **bvpSolve** provides some efficient integration codes for the solution of Boundary Value Problems.

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

Acknowledgements
Work developed within the project "Numerical methods and software for differential equations".

REFERENCES

[1] U.M. Ascher, J. Christiansen, R.D. Russell, *Collocation software for boundary-value ODEs*, Acm Trans. Math. Software **7** (1981), 209–222.

[2] U.M. Ascher, R.M.M. Mattheij, R.D. Russell, *Numerical solution of boundary value problems for ordinary differential equations*, vol. 13 of *Classics in Applied Mathematics*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995, ISBN 0-89871-354-4. Corrected reprint of the 1988 original.

[3] U.M. Ascher, L.R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM, Philadelphia, 1998.

[4] U.M. Ascher, R.J. Spiteri, *Collocation software for boundary value differential-algebraic equations*, SIAM J. Sci. Comput. **15** (1994) 4, 938–952.

[5] G. Bader, U.M. Ascher, *A new basis implementation for a mixed order boundary value ODE solver*, SIAM Journal on Scientific and Statistical Computing **8** (1987), 483–500.

[6] Z. Bashir-Ali, J.R. Cash, H.H.M. Silva, *Lobatto deferred correction for stiff two-point boundary value problems*, Comput. Math. Appl. **36** (1998) 10–12, 59–69, Advances in difference equations, II.

[7] L. Brugnano, F. Mazzia, D. Trigiante, *Fifty Years of Stiffness*, [in:] Recent Advances in Computational and Applied Mathematics, Springer Science+Business Media B.V., 2011.

[8] L. Brugnano, D. Trigiante, *On the characterization of stiffness for ODEs*, Dynam. Contin. Discrete Impuls. Systems **2** (1996) 3, 317–335.

[9] J.R. Cash, *Algorithms for the solution of two-point boundary value problems*, `http://www2.imperial.ac.uk~jcash/BVP_software/readme.php`.

[10] J.R. Cash, D. Hollevoet, F. Mazzia, A.M. Nagy, *Algorithm 927: the MATLAB code* `bvptwp.m` *for the numerical solution of two point boundary value problems*, ACM Trans. Math. Software **39** (2013) 2, Art. 15, 12.

[11] J.R. Cash, F. Mazzia, *A new mesh selection algorithm, based on conditioning, for two-point boundary value codes*, J. Comput. Appl. Math. **184** (2005) 2, 362–381.

[12] J.R. Cash, F. Mazzia, *Hybrid mesh selection algorithms based on conditioning for two-point boundary value problems*, JNAIAM J. Numer. Anal. Ind. Appl. Math. **1** (1) (2006), 81–90.

[13] J.R. Cash, F. Mazzia, *Conditioning and hybrid mesh selection algorithms for two-point boundary value problems*, Scalable Computing: Practice and Experience **10(4)** (2009), 347–361.

[14] J.R. Cash, G. Moore, R. Wright, *An automatic continuation strategy for the solution of singularly perturbed nonlinear boundary value problems*, ACM Transaction of Mathematical Software **27** (2) (2001), 245–266.

[15] J.R. Cash, M.H. Wright, *A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation.*, SIAM J. Sci. Statist. Comput. **12** (4) (1991), 971–989.

[16] J.W. Eaton, *GNU Octave Manual*, Network Theory Limited, 2002, ISBN 0-9541617-2-6.

[17] W.E. Enright, P.H. Muir, *Runge-Kutta software with defect control for boundary value odes*, SIAM J. Sci. Comput. **17** (1996), 479–497.

[18] W.H. Enright, P.H. Muir, *Efficient classes of Runge-Kutta methods for two-point boundary value problems*, Computing **37** (4) (1986), 315–334.

[19] E. Hairer, S.P. Norsett, G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems.* 2nd revised ed., Springer-Verlag, Heidelberg, 2009.

[20] E. Hairer, G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems.*, Springer-Verlag, Heidelberg, 1996.

[21] F. Iavernaro, F. Mazzia, D. Trigiante, *Stability and conditioning in numerical analysis*, JNAIAM J. Numer. Anal. Ind. Appl. Math. **1** (1) (2006), 91–112.

[22] The Mathworks, *Matlab release 2011b.*

[23] F. Mazzia, A.M. Nagy, *Stiffness detection strategy for explicit Runge Kutta methods*, AIP Conference Proceedings **1281** (2010) 1, 239–242.

[24] F. Mazzia, A. Sestini, D. Trigiante, *B-spline linear multistep methods and their continuous extensions*, SIAM J. Numer. Anal. **44** (2006) 5, 1954–1973 (electronic).

[25] F. Mazzia, A. Sestini, D. Trigiante., *The continous extension of the B-spline linear multistep metods for BVPs on non-uniform meshes.*, Appl. Numer. Math. **59** (2009) 3–4, 723–738.

[26] F. Mazzia, D. Trigiante, *A hybrid mesh selection strategy based on conditioning for boundary value ODE problems*, Numer. Algorithms **36** (2004) 2, 169–187.

[27] F. Mazzia, D. Trigiante, *Efficient strategies for solving nonlinear problems in BVPs codes*, Nonlinear Stud. **17** (2010) 4, 309–326.

[28] M.B. Monagan, K.O. Geddes, K.M. Heal, G. Labahn, S.M. Vorkoetter, J. McCarron, P. DeMarco, *Maple 10 Programming Guide*, Maplesoft, Waterloo ON, Canada, 2005.

[29] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2011.

[30] Scilab Consortium, *Scilab: The free software for numerical computation*, Scilab Consortium, Digiteo, Paris, France, 2011.

[31] K. Soetaert, **rootSolve***: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations*, 2009. R package version 1.6.

[32] K. Soetaert, J.R. Cash, F. Mazzia, *Solving Differential Equations in R*, Springer, 2012, ISBN 978-3-642-28069-6.

[33] K. Soetaert, J.R. Cash, F. Mazzia, **bvpSolve***: Solvers for Boundary Value Problems of Ordinary Differential Equations*, 2013, R package version 1.2.4.

[34] K. Soetaert, J.R. Cash, F. Mazzia, **deTestSet***: Testset for differential equations*, 2013, R package version 1.1.1.

[35] K. Soetaert, F. Meysman, *Reactive transport in aquatic ecosystems: Rapid model prototyping in the open source software R*, Environmental Modelling and Software, in press, (2011).

[36] K. Soetaert, T. Petzoldt, R.W. Setzer, *R-package* **deSolve***, Writing Code in Compiled Languages*, 2009, Package vignette.

[37] K. Soetaert, T. Petzoldt, R.W. Setzer, *Solving differential equations in R: Package* **deSolve**, Journal of Statistical Software **33** (2010) 9, 1–25.

[38] S. Theußl, A. Zeileis, *Collaborative Software Development Using R-Forge*, The R Journal **1** (2009) 1, 9–14.

[39] I. Wolfram Research, *Mathematica Edition, Version 8.0*, Wolfram Research Inc., 2010.

Francesca Mazzia
francesca.mazzia@uniba.it

Università degli Studi di Bari
Dipartimento di Matematica
Via Orabona 4, 70125 Bari, Italy

Jeff R. Cash
j.cash@imperial.ac.uk

Imperial College London
South Kensington Campus
Department of Mathematics
London SW7 2AZ, United Kingdom

Karline Soetaert
karline.soetaert@nioz.nl

Royal Netherlands Institute of Sea Research (NIOZ)
4401 NT Yerseke, The Netherlands

Received: October 4, 2013.
Accepted: November 21, 2013.