

IMPLEMENTATION OF A SOLUTION ROUTINE FOR SINGULAR BOUNDARY VALUE PROBLEMS

WINFRIED AUZINGER
GÜNTER KNEISL
OTHMAR KOCH
EWA B. WEINMÜLLER

ANUM PREPRINT No. 24/01



TECHNISCHE
UNIVERSITÄT
WIEN
VIENNA
UNIVERSITY OF
TECHNOLOGY

INSTITUTE FOR APPLIED MATHEMATICS
AND NUMERICAL ANALYSIS

IMPLEMENTATION OF A SOLUTION ROUTINE FOR SINGULAR BOUNDARY VALUE PROBLEMS

G. Kneisl

in cooperation with

W. Auzinger, O. Koch, W. Polster and E. Weinmüller

Abstract

This paper deals with the implementation of a MATLAB-solver for singular boundary value problems with a singularity of the first kind. This solver includes a mesh adaptation routine that selects meshes according to the local smoothness of the solution rather than to the smoothness of the direction field. The mesh selection is based on a new global error estimator for collocation schemes, which has recently been developed at the Institute for Applied Mathematics and Numerical Analysis and proves robust with respect to the singularity.

1 Introduction

1.1 Problem statement

We consider the boundary value problem for a system of *singular* ODEs

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) := \frac{1}{(t-a)} M(t) \cdot \mathbf{y}(t) + \mathbf{g}(t, \mathbf{y}(t)), \quad t \in (a, b), \quad (1a)$$

$$\mathbf{R}(\mathbf{y}(a), \mathbf{y}(b)) = \mathbf{0}, \quad (1b)$$

where M is a matrix which depends continuously on t and \mathbf{y} , \mathbf{g} and \mathbf{R} are smooth vector-valued functions. Note that a problem of this form is well-posed only under certain restrictions concerning the function \mathbf{R} . We attempt to find an approximation of the (locally) unique solution $\mathbf{y}^*(t)$ that satisfies a prescribed tolerance with as little computational effort as possible. This requires the use of an adaptive mesh selection strategy based on error estimation.

1.2 Motivation

Mathematical models of numerous applications from physics and chemistry take the form of systems of time-dependent partial differential equations subject to initial-boundary conditions, e.g. Ginzburg-Landau equations which arise in some physical contexts such as ferromagnetic systems, superconductivity models, models for unidirectional ring lasers and laser hydrodynamics.

For the investigation of stationary solutions many of these models can be reduced to singular systems of ordinary differential equations, especially when – due to

symmetries in the geometry of the problem and the problem data – polar, cylindrical or spherical coordinates can be used.

Numerical computation of homoclinic and heteroclinic orbits is of interest in describing structural changes in dynamical systems. Such problems take the form of a boundary value problem posed on an infinite interval. The computation based on the arclength parametrization of the orbit, however, uses a finite interval and can be treated with the techniques developed for singular problems.

Although there are various solution approaches suited for particular applications, there is no professional software for (at least wide subclasses of) singular problems, so standard codes (e.g. COLSYS, see [1]) are often used to solve singular problems numerically. If the underlying method does not involve function evaluations at the singular point, such a code will in principle be able to solve the problem¹. Unfortunately, such standard codes work very inefficiently in the region close to the singularity, even if the solution is very smooth. The local error estimation procedures tend to “recognize” that the direction field close to the singular point is very unsmooth and choose unnecessarily fine meshes, disregarding the smoothness of the solution and its derivatives.

1.3 Solution approach

We decided to use collocation for the numerical solution of the underlying boundary value problems. A collocating solution is a piecewise polynomial function which satisfies the given ODE at a finite number of nodes (collocation points). This approach shows advantageous convergence properties compared to other direct higher order methods (see [6], [12]), which may show order reductions and become inefficient in the presence of a singularity, see for example [7].

Furthermore, we decided to control the global error instead of monitoring the local error because of the unsmoothness of the latter near the singular point and order reductions it suffers from, cf. [5], [9]. The implemented error estimator was proposed in its classical version by Stetter [11] and is based on an idea due to Zadunaisky [13], originally formulated for solutions obtained by Runge-Kutta schemes. The same idea is also the basis for the acceleration technique known as *Iterated Defect Correction*, cf. [4], [8]. The classical error estimate works satisfactorily in the mesh points (i.e. when the information in the collocation points is ignored) but fails to be asymptotically correct for finer grids including collocation points. Modifications proposed in [3] remedy this problem and provide an asymptotically correct error estimate for the full grid, enhancing the efficiency of the estimate and the grid flexibility.

The mesh selection strategy is based on equidistribution of the global error. A detailed description is given in [10].

¹The new MATLAB 6 code `bvp4c` is based on a Lobatto scheme including function evaluations at the endpoints of the interval and therefore cannot be directly used for singular systems.

2 The solution routines

2.1 Modules

The solver package consists of the files

<code>bvpadmsh.m</code>	driver routine including mesh adaption
<code>bvpcol.m</code>	collocation solver
<code>errest.m</code>	error estimator
<code>bvpset.m</code>	tool for setting solution options
<code>optimset.m</code>	tool for setting zerofinder options (provided by MATLAB.)
<code>bvpplot.m</code>	output routine
<code>bvpphas2.m</code>	output routine
<code>bvpphas3.m</code>	output routine
<code>demofile1.m</code>	<code>bvpfile</code> that demonstrates how to ...
<code>demofile2.m</code>	... define BVPs
<code>demofile3.m</code>	... use parameters
<code>demofile4.m</code>	... set options
<code>demofile5.m</code>	... define multidimensional problems
<code>demofile6.m</code>	... set zerofinder options
<code>demofile7.m</code>	... use output functions
<code>demofile8.m</code>	... trace the iteration

2.2 Solver syntax

From the MATLAB command line or any MATLAB program, `bvpadmsh` is called by

```
[tau,y]=bvpadmsh(bvpfile[,tau0,y0,bvptopt,param1,param2,...])
```

where

- `bvpfile` is the name of an m-file that defines the right hand side of the differential equation, the boundary conditions and some Jacobians. `bvpfile` is discussed in Sec. 2.3.
- `tau0` is a strictly monotonous sequence that defines an initial guess for the mesh. If `tau0` is not specified, `bvpadmsh` evaluates

```
bvpfile('tau0')
```

to obtain it and selects a suitable mesh automatically if that fails.

- `y0` is an initial approximation of the solution and should be a $d \times (N + 1)$ -matrix, where d is the dimension of the system to be solved and $N + 1$ is the number of mesh points in `tau0`. If `y0` is not specified, `bvpadmsh` evaluates

```
bvpfile('y0',tau0)
```

to obtain it and uses the default initial approximation (all zeros) if that fails. For linear problems, no initial approximation is necessary.

- `bvpopt` is a parameter struct that determines the choice of collocation points, the basis for the space of collocation polynomials, the tolerance for the zero-finder, etc. `bvpopt` should be generated by `bvpset`. If no options are specified, `bvpdmesh` evaluates

```
bvpfile('bvpopt')
```

to obtain them and uses the default options if that fails.

- `param1`, `param2` are parameters of the boundary value problem that are passed on to `bvpfile` (see the header of `BVPFilePattern` in Sec. 2.3).
- `tau` is the final mesh produced by `bvpdmesh`.
- `y` is the final solution approximation on the mesh `tau`.

2.3 The `bvpfile`

A `bvpfile` is a MATLAB m-file that defines the boundary value problem to be solved. The generic `bvpfile` reads

```
function out=BVPFilePattern(flag,t,y,ya,yb,param1,param2,...)

switch flag
case 'f'           % rhs of the differential equation
    out=<insert f(t,y) here>;
case 'df/dy'      % Jacobian of the rhs
    out=<insert df/dy(t,y) here>;
case 'R'          % boundary condition
    out=<R(ya,yb)>;
case 'dR/dya'     % Jacobian of the boundary condition wrt. ya
    out=<dR/dya(ya,yb)>;
case 'dR/dyb'     % Jacobian of the boundary condition wrt. yb
    out=<dR/dyb(ya,yb)>;
case 'tau0'       % (optional)
    out=<initial mesh>;
case 'y0'         % (optional)
    out=<initial approximation y0(t)>
case 'bvpopt'     % (optional)
    out=<solution options for the BVP>
otherwise
    error('unknown flag');
end
```

To specify a boundary value problem, one should simply provide explicit expressions for the terms enclosed by `<...>`.

2.4 Examples

We demonstrate the performance of the solver using the following singular boundary value problems:

$$\mathbf{y}' = \frac{1}{t} \begin{pmatrix} 0 & 1 \\ 0 & -1 \end{pmatrix} \mathbf{y} - \begin{pmatrix} 0 \\ ty_1^5 \end{pmatrix}, \quad (2a)$$

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{y}(0) + \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \mathbf{y}(1) - \begin{pmatrix} \sqrt{3}/2 \\ 0 \end{pmatrix} = \mathbf{0}, \quad (2b)$$

$$\mathbf{y}' = \frac{1}{t} \begin{pmatrix} 0 & 1 \\ 2 & 6 \end{pmatrix} \mathbf{y} + \begin{pmatrix} 0 \\ t(-4\beta^4 t^4 \sin(\beta^2 t^2) - 10 \sin(\beta^2 t^2)) \end{pmatrix}, \quad (3a)$$

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \mathbf{y}(0) + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \mathbf{y}(1) - \begin{pmatrix} 0 \\ \sin(\beta^2) \end{pmatrix} = \mathbf{0}, \quad (3b)$$

$$\mathbf{y}' = \frac{1}{t} \begin{pmatrix} 0 & 1 \\ 1 + \alpha^2 t & 0 \end{pmatrix} \mathbf{y} + \begin{pmatrix} 0 \\ ct^{k-1} e^{-\alpha t} (k^2 - 1 - \alpha t(1 + 2k)) \end{pmatrix}, \quad (4a)$$

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \mathbf{y}(0) + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \mathbf{y}(1) - \begin{pmatrix} 0 \\ ce^{-\alpha} \end{pmatrix} = \mathbf{0}, \quad (4b)$$

where $\beta = 5$, $\alpha = 40$, $k = 36$ and $c = (\frac{\alpha}{k})^k e^k$. The solution of (2) is very smooth along the integration interval, the solutions of (3) and (4) are strongly varying close to and far away from the singular point, respectively. The meshes and solution approximations produced by `bvpdmesh` are shown in Fig. 1.

3 Developing the solver

3.1 Efficient referencing

To minimize the run-time of a program such as `bvpdmesh.m`, one has to know about MATLAB's internal array handling.

There is little to say about one-dimensional arrays. No matter if they represent row or column vectors, the elements of a vector are always stored in linear succession in the computer's memory:

array	memory
$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$	$a_1 \ a_2$
(a_1, a_2)	$a_1 \ a_2$

Two-dimensional arrays are more interesting, as they are stored *columnwise*,

array	memory
$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$	$a_{11} \ a_{21} \ a_{12} \ a_{22}$

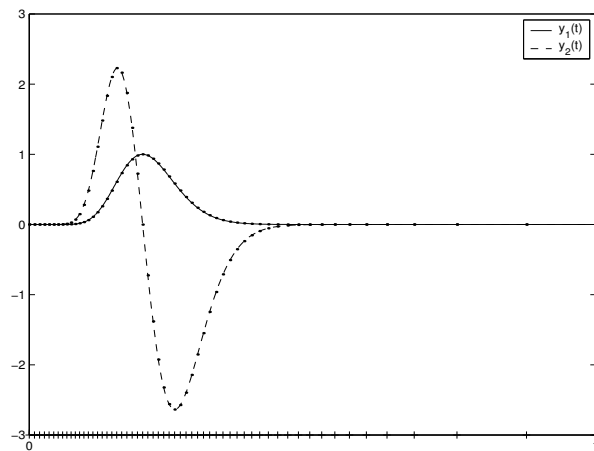
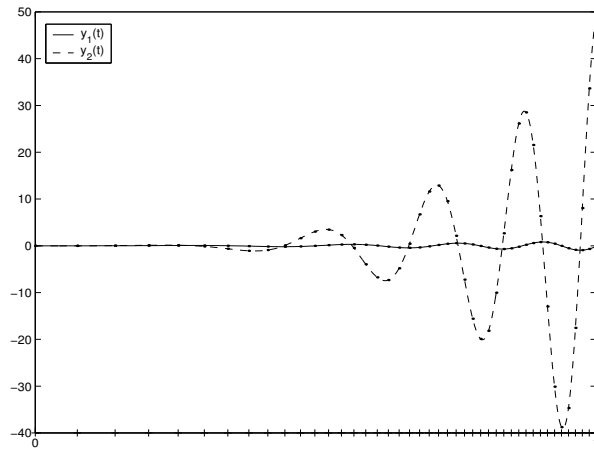
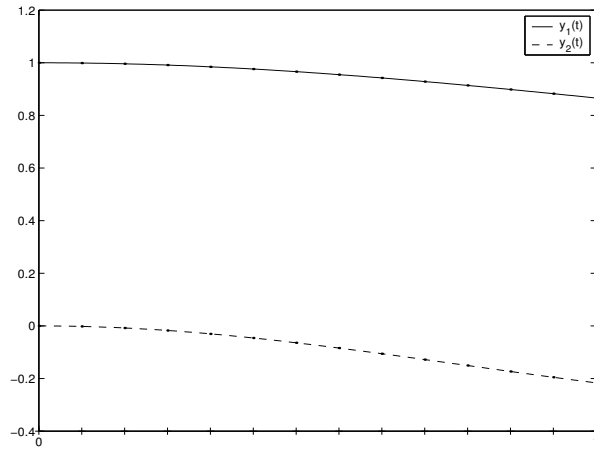


Figure 1: Mesh and solution of (2), (3) and (4)

and hence referencing rows and columns differs significantly. Extracting a column of a matrix and assigning it to another variable means copying a *connected memory area*, whereas for referencing a row, the appropriate elements have to be singled out first. Referencing columns is thus faster than referencing rows. Three-dimensional arrays $A(1:k, 1:l, 1:m)$ are represented in memory as

$$a_{111} \cdots a_{k11} \ a_{121} \cdots a_{kl1} \ a_{112} \cdots a_{kl2} \cdots a_{klm},$$

therefore referencing subarrays (matrices)

$$A(:, :, j)$$

is fastest.

3.2 Vectorizing

Vectorizing is another technique of reducing the run time of a MATLAB program. The credo of vectorizing is

Use as few loops as possible !

So if there exists a MATLAB command that performs the task of a loop, it should be used instead. As an example, we discuss a vectorized way to assemble a matrix

$$B = (\underbrace{A \ A \ \dots \ A}_{m \text{ times}})$$

from a given $n \times n$ matrix A . The source code of the obvious loop version reads

```
B=zeros(m*n,n); % preallocation of memory
for k=1:m
    B(:,(k-1)*m+1:k*m) = A;
end
```

but we can do better than that. First arrange the elements of A in linear succession,

$$\mathbf{a} = A(:)$$

then use

$$\mathbf{b} = \mathbf{a} * \mathbf{ones}(1,m)$$

to build

$$b = \begin{pmatrix} a_{11} & \cdots & a_{11} \\ a_{21} & \cdots & a_{21} \\ \vdots & & \vdots \\ a_{nn} & \cdots & a_{nn} \end{pmatrix}.$$

The linear memory representations of \mathbf{b} and B already coincide, only the shape is different. Reshaping \mathbf{b} yields the desired result,

$$B = \mathbf{reshape}(\mathbf{b}, n, n*m)$$

Combining all these steps, we obtain

$$B = \mathbf{reshape}(A(:)*\mathbf{ones}(1,m), n, n*m)$$

straightforward loop	100 %
lo-hi loop	83 %
idx loop	61 %
vectorized	31 %

Table 1: Relative run-times

3.3 Indexing

Sometimes, vectorized alternatives to loops are not available or would ruin the readability of a code segment. In such a case, one should at least try to avoid expensive calculations due to indexing, especially if the same indices are used several times. To the above example, such a loop alternative could read

```

idx = reshape(1:n*m,n,m); % assemble indices
B=zeros(m*n,n);          % preallocation of memory
for k=1:m
    B(:,idx(:,m)) = A;
end

```

This method of indexing is quite fast, but it is also quite memory consuming. In fact, the index vector `idx` needs the same amount of memory as `B`. A less expensive alternative would be

```

lo = 1:n:n*m;
hi = n:n:n*m; % hi = lo + (n-1)
B=zeros(m*n,n); % preallocation of memory
for k=1:m
    B(:,lo(m):hi(m)) = A;
end

```

Table 1 shows the relative execution times of all the discussed alternatives compared to the straightforward loop implementation ($n = 5$, $m = 10$).

3.4 A selected subroutine from the solution package

The short subroutine described below serves to illustrate the considerations of the previous sections by means of a practical example and demonstrates that even complicated looking tasks may allow a vectorized implementation.

The *monitor vector* $\theta \in \mathbb{R}^{1 \times N+1}$ is used in the mesh adaption process to measure the global error in a way suitable for equidistribution. From θ , the averaged $\bar{\theta} \in \mathbb{R}^{1 \times N+1}$ defined by

$$\bar{\theta}_k = \frac{1}{\min(N+1, k+s) - \max(1, k-s) + 1} \sum_{\ell=\max(1, k-s)}^{\min(N+1, k+s)} \theta_\ell$$

is to be calculated in a vectorized way. This seems to be a difficult task at the first glance, but it has a very elegant solution.

We generate the matrix $A \in \mathbb{R}^{2s+1 \times (N+1)+2s+1}$

$$A = \begin{pmatrix} \theta_1 & \theta_2 & \cdots & & \theta_{N+1} & 0 & 0 & \cdots & 0 \\ 0 & \theta_1 & \theta_2 & \cdots & & \theta_{N+1} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & & & & \ddots & \vdots \\ 0 & \cdots & 0 & \theta_1 & \theta_2 & \cdots & & \theta_{N+1} & 0 \\ 0 & \cdots & 0 & 0 & \theta_1 & \theta_2 & \cdots & & \theta_{N+1} \end{pmatrix},$$

extract the submatrix $A(:, (s+1) : (N+s+1))$ and take the sums along the columns,

```
a = [theta zeros(1,2*s+1)]' * ones(1,2*s+1);
A = reshape( a(1:(2*s+1)*(N+2*s+1)), N+2*s+1, 2*s+1)';

b = sum(A(:,s+1:N+s+1));
```

Then we divide by the number of relevant elements to obtain $\bar{\theta}$,

```
n = [(s+1):(2*s+1) (2*s+1)*ones(1,N-(2*s+1)) (2*s+1):-1:(s+1)];

theta_bar = b./n;
```

For the typical values $N = 100$, $s = 5$, execution time is reduced by 81% compared to the straightforward loop version

```
theta_bar = zeros(1,N+1);
for i=1:s+1
    theta_bar(i) = sum(theta(1:s+i)) / (s+i);
    theta_bar(end-i+1) = sum(theta(end-(s+i)+1:end)) / (s+i);
end

two_s1 = 2*s+1;
for i=s+2:(N+1)-(s+1)
    theta_bar(i) = sum(theta(i-s:i+s)) / two_s1;
end
```

References

- [1] U. ASCHER, J. CHRISTIANSEN, AND R. RUSSEL, *A collocation solver for mixed order systems of boundary values problems*, Math. Comp., 33 (1978), pp. 659–679.
- [2] U. ASCHER, R. MATTHEIJ, AND R. RUSSELL, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [3] W. AUZINGER, O. KOCH, AND E. WEINMÜLLER, *Efficient collocation schemes for singular boundary value problems*. Submitted to Numer. Algorithms.
- [4] R. FRANK, *The method of Iterated Defect Correction and its application to two-point boundary value problems, Part I*, Numer. Math., 25 (1976), pp. 409–419.

- [5] M. GRÄFF AND E. WEINMÜLLER, *Schätzungen des lokalen Diskretisierungsfehlers bei singulären Anfangswertproblemen*, Techn. Rep. Nr. 66/86, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technolgy, Austria, 1986.
- [6] F. D. HOOG AND R. WEISS, *Collocation methods for singular boundary value problems*, SIAM J. Numer. Anal., 15 (1978), pp. 198–217.
- [7] ———, *The application of Runge-Kutta schemes to singular initial value problems*, Math. Comp., 44 (1985), pp. 93–103.
- [8] O. KOCH AND E. WEINMÜLLER, *Iterated Defect Correction for the solution of singular initial value problems*. To appear in SIAM J. Numer. Anal.
- [9] P. KOFLER, *Theorie und numerische Lösung singulärer Anfangswertprobleme gewöhnlicher Differentialgleichungen mit der Singularität erster Art*, Ph. D. Thesis, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 1998.
- [10] W. POLSTER, Master Thesis, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 2001.
- [11] H. J. STETTER, *The defect correction principle and discretization methods*, Numer. Math., 29 (1978), pp. 425–443.
- [12] E. WEINMÜLLER, *On the numerical solution of singular boundary value problems of second order by a difference method*, Math. Comp., 46 (1986), pp. 93–117.
- [13] P. ZADUNAISKY, *On the estimation of errors propagated in the numerical integration of ODEs*, Numer. Math., 27 (1976), pp. 21–39.