BVPTWP Manual

J. R. CASH*, D. HOLLEVOET⋆, F. MAZZIA†, A. M. NAGY‡

*Department of Mathematics, Imperial College, South Kensington, London SW7, England.
e-mail: j.cash@imperial.ac.uk

⋆Vakgroep Toegepaste Wiskunde en Informatica, Universiteit Gent, Krijgslaan 281 S9, 9000 Gent.
e-mail: davy.hollevoet@ugent.be

†Dipartimento di Matematica, Università di Bari, Via Orabona 4, 70125 Bari (Italy).
e-mail: mazzia@dm.uniba.it

‡Dipartimento di Matematica, Università di Bari, Via Orabona 4, 70125 Bari (Italy)
Department of Mathematics, Benha University, 13518 Benha (Egypt).
e-mail: abdelhameed_nagy@yahoo.com

Contents

1 Introduction

In recent years much attention has been given to the numerical solution of boundary value problems in ODEs. Of particular interest has been the solution of singularly perturbed problems. This type of problem arises in various fields of science and engineering such as fluid mechanics, physics, chemistry, mechanics, chemical reactor theory, convection diffusion processes, optimal control and other branches of applied mathematics. Singular perturbation problems depend on the presence of a small, positive parameter which provides a multi-scale character to the solution. That is there are layer(s) where the solution varies very rapidly in some parts of the region of integration and varies slowly in some other parts. `bvptwp` is a MATLAB program that calculates an approximate solution for two-point Boundary Value Problems that may or may not be singularly perturbed. The general BVP is of the form:

$$y' = f(x, y), \quad a \le x \le b \tag{1}$$

$$g(y(a), y(b)) = 0, \tag{2}$$

where $g = (g_1, g_2, \cdots, g_n)^T$ is a vector functions. The functions f and g_i are assumed to be differentiable. The problem must be posed as a first-order system. This requirement is not unduly restrictive, however, since standard techniques can be used to convert an nth-order equation to a system of n first-order equations. For example, the second-order singularly perturbed problem

$$\lambda y'' = f(x, y, y'), \quad 0 < x < 1, \quad y(0) = \alpha, \quad y(1) = \beta, \tag{3}$$

where $0 < \lambda \ll 1$, $x \in \mathbb{R}$, can be converted to the following first-order system:

$$\begin{aligned} u' &= z, \\ z' &= \frac{1}{\lambda} f(x, u, z), \\ u(0) &= \alpha, \quad u(1) = \beta. \end{aligned}$$

2 BVPTWP Package

The purpose of this section is to give a brief introduction to the use of the routines in Matlab by means of some fairly simple examples. In particular, we show how to compute an approximate solution and how to evaluate these solutions by various graphical tools. Although the examples given below do not touch upon all the features of `bvptwp` tools, they illustrate the fundamental ideas underlying the package.

2.1 Installation

Extract the contents of the archive `bvptwp_1.0.zip` into a directory of your choice, e.g. `bvptwp-path`. This newly created directory should now be added to the MATLAB path. This can be done temporarly (for one MATLAB session) by executing

```
>> addpath ('bvptwp-path/bvptwp_1.0')
```

on the MATLAB command line (be sure to use single quotes). The package can also be added to the MATLAB path permanently via `File/Set Path...` and `Add Folder...`. The `bvptwp` package is now ready for use.

2.2 How to solve a problem using bvptwp

The solver `bvptwp` is an interface for solving BVPs by using a deferred correction scheme based on LOBATTO methods (`twpbvp_l`), a deferred correction scheme based on MIRK methods (`twpbvp_m`) or a continuation strategy based on LOBATTO methods (`acdc`). The interface is built such that it can be used in a MATLAB environment like the well-known solvers `bvp4c` and `bvp5c`. The three codes can also be used with a hybrid mesh selection strategy based on conditioning by means of the corresponding variants `twpbvpc_l`, `twpbvpc_m` and `acdcc`.

To solve a problem with m components, `bvptwp.m` can be invoked with three parameters:

```
>> SOL = bvptwp(odefun, bcfun, solinit)
```

or four parameters:

```
>> SOL = bvptwp(odefun, bcfun, solinit, options)
```

- `odefun`: function handle
  The provided function implements $f(x, y)$ of the differential equation. If option `Vectorized` is not enabled, this function is given a scalar x and a vector y and should return a $m \times 1$ matrix.
- `bcfun`: function handle
  The provided function computes $bc(y(a), y(b))$ at the boundaries.
- `solinit`: struct
  A compatible structure for `solinit` can be constructed with the function `bvpinit` from MATLAB. Afterwards, an additional field `fixpnt` can be added to the structure obtained, which specifies additional mesh points in which the solution should be calculated.
- *(Optional)* `options`: struct
  A structure that specifies options for the solver. Can be constructed with `bvptwpset`.

The output `SOL` is a structure with

- `SOL.x`: last mesh selected by `bvptwp`.
- `SOL.y`: approximation to $y(x)$ at the mesh points of `SOL.x`.
- `SOL.solver`: `'twpbvp_m'` or `'twpbvp_l'` or `'acdc'` or `'twpbvpc_m'` or `'twpbvpc_l'` or `'acdcc'`.
- `SOL.lambda`: the final value of the continuation parameter used (only for `acdc`, `acdcc`).
- `SOL.iflbvp`:
  - 0 : the code solved the problem.
  - −1 : the code solved a problem with a different continuation parameter (only for `acdc`, `acdcc`).
  - 1 : tha code failed (maximum number of meshpoints reached).

2 : the code failed (maximum number of possible meshes reached, default 100, only for `twpbvpc_m`, `twpbvpc_l`).

3 : the code failed (maximum number of continuation steps reached, only for `acdc`, `acdcc`).

4 : the code failed (unknown error).

- `SOL.condpar`: information about the conditioning parameters (only for `twpbvpc_m`, `twpbvpc_l`, `acdcc`)
  - `SOL.condpar.kappa`: conditioning of the bvp in inf norm.
  - `SOL.condpar.kappa1`: conditioning related to changes in the initial values (inf norm).
  - `SOL.condpar.kappa2`: conditioning related to the Green's function (inf norm).
  - `SOL.condpar.gamma1`: conditioning related to changes in the initial values (1 norm).
  - `SOL.condpar.sigma`: stiffness parameter.
  - `SOL.condpar.stabcond`: 1 or 0 if the conditioning parameters stabilized.
- `SOL.stats`: Computational cost, statistics and information about the maximum scaled error computed.

```
>> SOL = bvptwp(odefun, bcfun, solinit, options)
```

solves as above with default parameters replaced by values in `options`, a structure created with the `bvptwpset` function. To reduce the run time greatly, use `options` to supply a function for evaluating the Jacobian and/or Jvectorize.

2.2.1 Options that can be specified in parameter `options` (defaults are marked with {}).

Options shared with `bvpc4.m` and `bvpc5.m`:

- `Vectorized`: `on|{off}`
  Set to `'on'` when `odefun` can be evaluated at several points at once. When enabled, the function handle is invoked as `odefun` $([x_1, \ldots, x_n], [y_1, \ldots, y_n])$ and is expected to return an $m \times n$ matrix.
- `FJacobian`: function handle `df`
  If provided, `df` implements the Jacobian $\partial f/\partial y$ of the problem. The function provided will be invoked as `df(x,y)` with scalar x and vector y. In the absence of this option, the Jacobian is calculated numerically with `odenumjac` from MATLAB.
- `BCJacobian`: function handle `dbc`
  A function handle that computes the partial derivatives $\partial bc/\partial y_a$ and $\partial bc/\partial y_b$ of the boundary conditions. Calls to this function will be made as `dbc(ya,yb)` and two $m \times m$ matrices are expected as output.
- `NMax`: positive integer $\{\lfloor 50000/m \rfloor\}$
  Maximum number of mesh points allowed.
- `RelTol`: positive scalar $\{1e\text{-}3\}$ or vector. Relative tolerance for the error. If `RelTol` is a vector a different tolerance is used for all components of the solution. The value 0 means that the corresponding component is not used in the error estimation.
- `Stats`: `on|{off}`
  Show a few statistics at the end of the computations.

  Additional options provided by `bvptwp.m`:

- `Solver:'twpbvp_m'|{'twpbvp_l'} (for linear problems) |'acdc'|{'twpbvpc_m'} (for non linear problems) |'twpbvpc_l' |'acdcc'`
  Use deferred correction based on MIRK (`twpbvp_m`) or LOBATTO methods (`twpbvp_l`) or a continuation strategy based on LOBATTO methods (`acdc`) or their implementation based on conditioning (`twpbvpc_m,twpbvpc_m,acdcc`).

- `LambdaStart`: positive scalar {0.5}
  Starting value for the continuation parameter, used only for `acdc(c)` solver.

- `LambdaMin`: positive scalar {1e-5}
  Final value for the continuation parameter, used only for `acdc(c)` solver.

- `JVectorized`: `on|{off}`
  Set to `'on'` if the function handle `df` specified for `FJacobian` is capable of computing the Jacobian at several points at once. If enabled, this setting will cause `df` to be called as `df`$([x_1, \ldots, x_n], [y_1, \ldots, y_n])$. This function should return a $m \times m \times n$ matrix.

- `Linear`: `on|{off}`
  Used to indicate whether or not the problem is linear. If enabled, the problem is solved taking into account the linear behavior.

- `MaxNumberOfMeshes`: positive integer {100}
  This is the maximum number of different meshes that could be used during computation. It is important to avoid loops.

- `MaxNumberOfContStep`: positive integer {150}
  This is the maximum number of continuation steps (only for `acdc/acdcc` solvers).

- `Debug`: `on|{off}`
  Enable or disable debugging information (very verbose!).

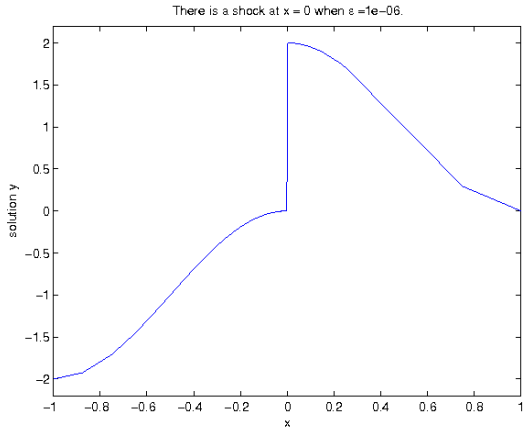

Figure 1: Output of the example shock_bvptwp.

2.3 Examples

2.3.1 Example 1

We consider the following singularly perturbed, linear two-point boundary value problem

$$\begin{aligned} &\epsilon y'' + xy' = -\epsilon\pi^2\cos(\pi x) - \pi x\sin(\pi x), \\ &y(-1) = -2, \quad y(1) = 0 \end{aligned} \tag{4}$$

whose exact solution is $y(x) = \cos(\pi x) + \mathrm{erf}(x/\sqrt{2\epsilon})/\mathrm{erf}(1/\sqrt{2\epsilon})$.

The following MATLAB function illustrates how `bvptwp.m` can be used to solve example (4).

```
function shock_bvptwp(solver)
%shock_bvptwp  The solution has a shock layer near x = 0
%   This is an example used in U. Ascher, R. Mattheij, and R. Russell,
%   Numerical Solution of Boundary Value Problems for Ordinary Differential
%   Equations, SIAM, Philadelphia, PA, 1995,  to illustrate the mesh
%   selection strategy of COLSYS.
%
%   For 0 < e << 1, the solution of
%
%       e*y'' + x*y' = -e*pi^2*cos(pi*x) - pi*x*sin(pi*x)
%
%   on the interval [-1,1] with boundary conditions y(-1) = -2 and y(1) = 0
%   has a rapid transition layer at x = 0.
%
%   For this problem,
%   analytical partial derivatives are easy to derive and the solver benefits
%   from using them.
%
%   By default, this example uses the 'twpbvpc_l' solver. Use syntax
%   SHOCK_BVPTWP(solver) to solve this problem with the another solver
%     available solvers are: 'twpbvp_m', 'twpbvpc_m', 'twpbvp_l', 'twpbvpc_l',
%                            'acdc', 'acdcc'
%
%   See also bvptwp, bvptwpset, bvptwpget, bvpinit, function_handle.
%   THIS MFILE IS ADAPDET FORM THE SHOCKBVP OF
%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2007 The MathWorks, Inc.
```

```
%   $Revision: 1.10.4.3 $  $Date: 2007/05/23 18:53:57 $

  if nargin < 1
    solver = 'twpbvpc_l';
  end


  % The differential equations written as a first order system and the
  % boundary conditions are coded in shockODE and shockBC, respectively. Their
  % partial derivatives are coded in shockJac and shockBCJac and passed to the
  % solver via the options. The option 'Vectorized' instructs the solver that
  % the differential equation function has been vectorized, i.e.
  % shockODE([x1 x2 ...],[y1 y2 ...]) returns [shockODE(x1,y1) shockODE(x2,y2) ...].
  % Such coding improves the solver performance.

  options = bvptwpset(   'FJacobian',@shockJac,'BCJacobian',@shockBCJac,...
                         'Solver',solver,'RelTol',[1e-4;0],'Linear','on');

  % A guess for the initial mesh and the solution
  %sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
  solinit =bvpinit(linspace(-1,1,9),zeros(1,2));

  e = 1e-6;
  if  strcmp(solver,'acdc')|| strcmp(solver,'acdcc')
      options=bvptwpset(options,'Lambdamin',e);
  end

  sol = bvptwp(@shockODE,@shockBC,solinit,options);

  % The final solution
  figure;
  plot(sol.x,sol.y(1,:));
  axis([-1 1 -2.2 2.2]);
  title(['There is a shock at x = 0 when \epsilon =' sprintf('%.e',e) '.']);
  xlabel('x');
  ylabel('solution y');

  % ------------------------------------------------------------------------
  % Nested functions -- e is shared with the outer function.
  %

  function dydx = shockODE(x,y,ExtraArgs)
    %SHOCKODE  Evaluate the ODE function (vectorized)
    if nargin==3
      e=ExtraArgs;
    end

    pix = pi*x;
    dydx = [   y(2,:)
               (-x.*y(2,:) - e*pi^2*cos(pix) - pix.*sin(pix))/e ];
  end
  % ------------------------------------------------------------------------

  function res = shockBC(ya,yb,ExtraArgs)
    %SHOCKBC  Evaluate the residual in the boundary conditions
    if nargin==3
      e=ExtraArgs;
    end
```

```
    res = [ ya(1)+2
            yb(1)   ];
  end
  %--------------------------------------------------------------------------

  function jac = shockJac(x,y,ExtraArgs)
    %SHOCKJAC  Evaluate the Jacobian of the ODE function
    %  x and y are required arguments.
    if nargin==3
      e=ExtraArgs;
    end
    jac = [ 0    1
            0 -x/e ];
  end
  %--------------------------------------------------------------------------

  function [dBCdya,dBCdyb] = shockBCJac(ya,yb,ExtraArgs)
    %SHOCKBCJAC  Evaluate the partial derivatives of the boundary conditions
    %  ya and yb are required arguments.
    if nargin==3
      e=ExtraArgs;
    end

    dBCdya = [ 1 0
               0 0 ];

    dBCdyb = [ 0 0
               1 0 ];
  end
end  % shock_bvptwp
```

2.3.2 Example 2

A second example compares the final mesh obtained from four solvers without continuation. The problem under consideration is

$$\begin{aligned} &y'' = \epsilon sinh(\epsilon y), \\ &y(0) = 0, \quad y(1) = 1. \end{aligned} \tag{5}$$

The following MATLAB function illustrates how `bvptwp.m` can be used to solve example (5).

```
function sinh_bvptwp()
% sinh_bvptwp
%  compares the final mesh obtained from four solvers without continuation.
% The problem under  consideration is
%    y'' = e sinh(e y),
%   with boundary conditions
%    y(0) = 0,     y(1)=1.
%
% For this problem:
%      Function f is vectorized
%      Analytical Jacobians are not implemented
%      RelTol is the same for both components
%      Printing statistics after solving is enabled
%
    options = bvptwpset('Vectorized','on',...
                'RelTol',1e-8,...
                'Linear','off',...
                'NMax',1000,...
                'Stats','on');

    % A guess for the initial mesh and the solution
    sol = bvpinit(linspace(0,1,18),[0,0]);

    e = 5.32675;

    % uses deferred correction with MIRK methods
    options = bvptwpset(options,'Solver','twpbvp_m');
    sol1 = bvptwp(@sinhODE,@sinhBC,sol,options);

    % uses deferred correction with MIRK methods and conditioning
    options = bvptwpset(options,'Solver','twpbvpc_m');
    sol2 = bvptwp(@sinhODE,@sinhBC,sol,options);

    % uses deferred correction with Lobatto methods
    options = bvptwpset(options,'Solver','twpbvp_l');
    sol3 = bvptwp(@sinhODE,@sinhBC,sol,options);

    % uses deferred correction with Lobatto methods and conditioning
    options = bvptwpset(options,'Solver','twpbvpc_l');
    sol4 = bvptwp(@sinhODE,@sinhBC,sol,options);



    %%% Nested functions, e is shared with the outer function

    function dydx = sinhODE(x,y)
        % Evaluate the ODE function (vectorized)
        dydx = [ y(2,:)
                 e*sinh(e*y(1,:)) ];
    end
```

```
    function res = sinhBC(ya,yb)
        % Evaluate the residual in the boundary conditions
        res = [ ya(1)
                yb(1)-1 ];
    end
end
```

The output is:

```
-----------------------------------------
Solver twpbvp_m.
The solution was obtained on a mesh of 42 points.
The maximum scaled error  is  6.897e+00.
There were 60 calls to the ODE function.
There were 23 calls to the BC function.
-----------------------------------------
Solver twpbvpc_m.
The solution was obtained on a mesh of 47 points.
The maximum scaled error  is  5.208e-01.
There were 60 calls to the ODE function.
There were 23 calls to the BC function.
-----------------------------------------
Solver twpbvp_l.
The solution was obtained on a mesh of 40 points.
The maximum scaled error  is  5.307e+00.
There were 60 calls to the ODE function.
There were 22 calls to the BC function.
-----------------------------------------
Solver twpbvpc_l.
The solution was obtained on a mesh of 41 points.
The maximum scaled error  is  3.515e+00.
There were 60 calls to the ODE function.
There were 22 calls to the BC function.
```