Institute of Analysis and Scientific Computing
Vienna University of Technology

BVPSUITE2.0

A MATLAB solver for singular BVPs in ODEs, EVPs and DAEs

Merlin Fallahpour, MSc.
Dr. Othmar Koch
Aron Sass, MSc.
Prof. Ewa B. Weinmüller

December 2019

Contents

1 Introduction **2**
1.1 Scope of `bvpsuite2.0` . . . 2
1.2 Solver . . . 2
1.3 About this manual . . . 2

2 The package **4**
2.1 Installation and first run . . . 4
2.2 Model problem . . . 5
2.3 Input . . . 6
2.3.1 Problem definition . . . 6
2.3.2 Solver settings . . . 8
2.3.3 Initial profile . . . 9
2.3.4 Pathfollowing struct . . . 10
2.4 Output . . . 10
2.5 Error estimation and mesh adaptation . . . 12

3 Specific problem definitions and examples **13**
3.1 Linear problem . . . 13
3.2 Non-linear problem . . . 14
3.3 Parameter-dependent problem . . . 18
3.4 Eigenvalue problem . . . 19
3.5 Index-1 DAE . . . 24
3.6 Problem on semi-infinite interval . . . 25
3.7 Pathfollowing . . . 27

A Boundary conditions inside the interval **35**

1 Introduction

1.1 Scope of `bvpsuite2.0`

The package `bvpsuite2.0` has been developed at the Institute for Analysis and Scientific Computing, Vienna University of Technology, and can be used for the numerical solution of implicit boundary value problems (BVPs) in ordinary differential equations (ODEs) as well as eigenvalue problems (EVPs), and Index-1 Differential-Algebraic Equations (DAEs). The ODE system can have a general implicit form and be of mixed order[1] subject to multi-point boundary conditions (BCs). Furthermore the interval can be finite or semi-infinite[2]. The BVP may also include unknown parameters to be calculated together with the unknown solution $\boldsymbol{z}$, in which case additional boundary conditions are required.
`bvpsuite2.0` is a product of years of research at the Institute for Analysis and Scientific Computing on the analysis and numerical solution of ODEs with time and space singularities. Through the years, the code evolved from `sbvp`, a collocation code for singular ODEs of the first order, to `bvpsuite1.1`, which can also handle explicit and implicit problems of arbitrary order, to `bvpsuite2.0`, which improves on usability and readability of the code. The code in the package is now structured in a simpler modular form.

1.2 Solver

As in the previous versions of the code, collocation is used for the numerical solution of the underlying boundary value problems. A collocation 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 [10]). However, the superconvergence in the context of collocation applied to singular problems can not be guaranteed in general, see for example [9] and [11]. Our estimate for the global error of the collocation solution is a classical error estimate based on mesh halving. To make the computations more efficient, an adaptive mesh selection strategy based on the control of residual and the global error estimate has been implemented. A detailed description of this mesh selection algorithm is given in [8].

1.3 About this manual

This manual aims to help users to get started with the computations of their own problems with `bvpsuite2.0`.
Starting in the next section, we will demonstrate, using a simple example, how to start using `bvpsuite2.0`. Then the model problem in a general form, for which `bvpsuite2.0`

[1] The highest involved derivative may vary with the solution component and it can also be zero, which means that algebraic constraint which do not involve derivatives are also admitted.

[2] $[a, \infty)$, where $a \geq 0$ and Dirichlet-type boundary conditions are posed at infinity

can find approximate solutions, is presented. Afterwards, the input arguments and also the output of the function call are briefly discussed.
Finally, in the third and final section, we present a few examples of BVPs to highlight some of the features of the seven modules of the code.

2 The package

2.1 Installation and first run

Create a directory for the `bvpsuite2.0`-package and unzip `bvpsuite2.0.zip` into this directory. The `bvpsuite2.0`-package contains a directory entitled `Examples` (these are treated one by one in Section 3), a license file, and furthermore the following files are included:

1	`bvpsuite2.m`	main function
2	`feval_problem.m`	access problem data
3	`coeffToValues.m`	transforms coefficients of polynomial in Runge-Kutta basis to values
4	`initial_coefficients.m`	transforms values to coefficients of polynomial in Runge-Kutta basis
5	`functionFDF.m`	returns system or Jacobian of system
6	`solveLinearProblem.m`	linear solver
7	`solveNonlinearProblem.m`	non-linear solver
8	`solve_nonlinear_sys.m`	non-linear solver iteration
9	`errorestimate.m`	error estimation procedure
10	`meshadaptation.m`	mesh adaptation procedure
11	`computeResidual.m`	used in `meshadaptation.m`
12	`computeEVPStart.m`	finds initial profiles of eigenvalues and eigenvectors
13	`pathfollowing.m`	pathfollowing routine
14	`dispDebug.m`	debugging tool
15	`getStandardCollocationPoints.m`	collocation points on $[0, 1]$
16	`trafo.m`	standard transformation of semi-infinite interval
17	`higherchainrule.m`	used in `trafo.m`
18	`template_bvp.m`	problem definition template
19	`default_settings.m`	solver settings template

More in-depth explanations and a short pseudo-code for these functions can be found in [4, Sec. 2.1].
Once all these files are in the folder, then add the directory to the MATLAB search path by using the function `addpath`, i.e.

```
addpath( 'The full path to the directory you created' );
```

into the command window.
You are now ready to use `bvpsuite2.0` and you can run `bvpsuite2.0` by calling

```
[x,y,s] = bvpsuite2( 'template_bvp', 'default_settings' );
```

in your command window.
The file `template_bvp.m` contains the problem definition and the file `default_settings.m` contains the solver settings for this problem. These two files are needed to call the function `bvpsuite2.m`. The file `template_bvp.m` describes the linear boundary value problem

$$z^{(3)}(t) = z''(t) - z'(t) + z(t), \quad t \in [-1, 1], \tag{1a}$$
$$z(-1) = \mathrm{e}^{-1}, z'(-1) = \mathrm{e}^{-1}, z''(-1) = \mathrm{e}^{-1}. \tag{1b}$$

In `default_settings.m` collocation with 3 Gaussian points is chosen to solve the problem on a starting grid with 51 mesh points. We prescribe the value 10^{-12} for the absolute and relative tolerance of the non-linear solver and 10^{-9} for the absolute and relative tolerance of the mesh adaptation.
When calling

```
plot( x, y );
figure; plot( s.x1tau, s.errest );
figure; plot( x, 0, 'o' );
```

a plot of the solution, a plot of the automatically generated error estimation and a plot of the distribution of the grid points will be displayed in this exact order.
Before going into further detail on how to use the different modules of `bvpsuite2.0`, we will present the general form of a model problem our code can handle.

2.2 Model problem

Let $\mathcal{I} = [a, b] \subset \mathbb{R}$ be given, as well as a vector $\boldsymbol{c} \in \mathbb{R}^q$ with $c_i \in \mathcal{I}$ and $c_i \neq c_j$ for $i \neq j$. Then, we try to find a vector function $\boldsymbol{z}(t) : \mathcal{I} \to \mathbb{R}^n$ and a vector of parameters $\boldsymbol{p} \in \mathbb{R}^s$, such that for all $t \in \mathcal{I}$ and $\lambda^* \in \mathbb{R}$ the ODE system

$$\boldsymbol{f}(t, \boldsymbol{p}, z_1(t), z_1'(t), \ldots, z_1^{(l_1)}(t), \ldots, z_i(t), \ldots, z_i^{(l_i)}(t), \ldots, z_n(t), \ldots, z_n^{(l_n)}(t), \lambda^*) = \boldsymbol{0}, \tag{2}$$

and the boundary conditions

$$\begin{aligned} &\boldsymbol{g}(\boldsymbol{p}, z_1(c_1), \ldots, z_1^{(l_1-1)}(c_1), \ldots, z_n(c_1), \ldots, z_n^{(l_n-1)}(c_1), \ldots \\ &\ldots, z_1(c_q), \ldots, z_1^{(l_1-1)}(c_q), \ldots, z_n(c_q), \ldots, z_n^{(l_n-1)}(c_q), \lambda^*) = \boldsymbol{0}, \end{aligned} \tag{3}$$

are satisfied. Here,

$$\boldsymbol{f} : \mathcal{I} \times \mathbb{R}^s \times \mathbb{R}^{\sum (l_i+1)} \times \mathbb{R} \to \mathbb{R}^n, \;\; \boldsymbol{g} : \mathbb{R}^s \times \mathbb{R}^{q \sum l_i} \times \mathbb{R} \to \mathbb{R}^{s + \sum l_i}$$

and we assume that $z_i \in C^{l_i - 1}(\mathcal{I})$ and $z_i^{(l_i)}$ exist, where l_i denotes the highest occurring derivative of z_i in (2). In the case of eigenvalue problems, (2) should be written as

$$\boldsymbol{f}(t, \boldsymbol{p}, z_1(t), z_1'(t), \ldots, z_1^{(l_1)}(t), \ldots, z_i(t), \ldots, z_i^{(l_i)}(t), \ldots, z_n(t), \ldots, z_n^{(l_n)}(t), \lambda^*) = \lambda \boldsymbol{z}(t),$$

with boundary conditions as in (3).
For a precise description of the collocation method used in `bvpsuite2.0`, we refer to [4, Sec. 1.2.1]. For the case of the specific treatment of eigenvalue problems, we refer to [3, Chap. 4]
In order to explain how to compute a numerical approximation to the solution to your own BVP with `bvpsuite2.0`, we will now look in detail at the input the function `bvpsuite2.m` requires. Afterwards the output it generates will be discussed.

2.3 Input

Begin by saving the files `template_bvp.m` and `default_settings.m` as e.g. `problem1.m` and `settings1.m` respectively in a new directory entitled `problem1dir` for instance. Now add the directory `problem1dir` to MATLAB's search path by using the function `addpath`, as it was used in Section 2.1.
The function `bvpsuite2.m` can be called with the following four arguments

```
[x,y,s] = bvpsuite2( problem, settings, initProfile, pathfoll );
```

To call the function `bvpsuite2` the arguments of the problem definition file and the solver settings file must be provided. `initProfile` and `pathfoll` are optional arguments that can be omitted, as shown in the first run in Section 2.1.
A description of these four input arguments is provided here.

2.3.1 Problem definition

The first input argument of `bvpsuite2` is the problem definition. The file `template_bvp.m` is a template which can be used to define the problem in the correct manner for `bvpsuite2.0` to process it.
In the model problem from Section 2.2, $z_i^{(j)}(t)$ is replaced by `z(i,j+1)`, p_i by `p(i)`, λ by `lambda`, λ^* by `lambda_p` and t by `t`, if they occur. Then in the problem definition, the following information can be provided:

- `'n'`: $\boldsymbol{n}$ Number of solution components;
- `'orders'`: $\boldsymbol{l}$ Row vector of the highest order of each component;
- `'problem'`: $\boldsymbol{f}$ Column vector of the left hand side of the implicit ODE system (2);
- `'jacobian'`: $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}$ $\boldsymbol{n} \times \boldsymbol{n} \times \max_{\alpha=1,\dots,n} \boldsymbol{l}_\alpha$-matrix, the Jacobi-matrices of $\boldsymbol{f}$ w.r.t. the solution components and their derivatives; `ret(i,j+1,k)` corresponds to $\frac{\partial f_i}{\partial z_j^{(k)}}$;
- `'interval'`: $[\mathbf{a}, \mathbf{b}]$ The interval on which the problem is solved;
- `'linear'`: `1` for a linear problem, `0` for a non-linear problem;
- `'parameters'`: s The number of unknown parameters;
- `'c'`: $\boldsymbol{c}$ Return a row vector of points in which the BCs are posed. If the BCs are only posed in $\mathbf{a}$ or in $\mathbf{b}$, then you may return an empty vector [];
- `'BV'`: $\boldsymbol{g}$ Column vector of the left hand side of (3), where **either** $\boldsymbol{c}$ was left empty, then replace $z_i^{(j)}(a)$ by `za(i,j+1)` and $z_i^{(j)}(b)$ by `zb(i,j+1)` **or** $\boldsymbol{c}$ was not left empty,

then replace $z_i^{(j)}(c_k)$ by `zc(i,j+1,k)`;

- `'dBV'`: $\frac{\partial \boldsymbol{g}}{\partial(\boldsymbol{z}(a),\boldsymbol{z}(b))}$ or $\frac{\partial \boldsymbol{g}}{\partial(\boldsymbol{z}(c_1),\ldots,\boldsymbol{z}(c_q))}$ The Jacobian of $\boldsymbol{g}$ w.r.t. $z_i^{(j)}(a)$ and $z_i^{(j)}(b)$, or $z_i^{(j)}(c_k)$; Here **either** `ret(1,i,j,k+1)` corresponds to $\frac{\partial g_i}{\partial z_j^{(k)}(a)}$ and `ret(2,i,j,k+1)` corresponds to $\frac{\partial g_i}{\partial z_j^{(k)}(b)}$, **or** `ret(cInd,i,j,k+1)` corresponds to $\frac{\partial g_i}{\partial z_j^{(k)}(c_{\mathrm{cInd}})}$;
- `'dP'`: $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}}$ The Jacobian of $\boldsymbol{f}$ w.r.t. p_i; `ret(i,j)` corresponds to $\frac{\partial f_i}{\partial p_j}$;
- `'dP_BV'`: $\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$ The Jacobian of $\boldsymbol{g}$ w.r.t. p_i; `ret(i,j)` corresponds to $\frac{\partial g_i}{\partial p_j}$;
- `'initProfile'`: The initial solution profile for the non-linear solver; `ret.initialMesh` shall be specified here as a row vector of length $t \geq 2$ and `ret.initialValues` as a $\boldsymbol{n} \times t$-matrix. If the BVP is parameter-dependent, then initial values for the parameters must also be provided `ret.parameters` as a row vector of length s;
- `'EVP'`: `1` for an eigenvalue problem, `0` else;
- `'dLambda'`: $\frac{\partial \boldsymbol{f}}{\partial \lambda}$ The Jacobian of $\boldsymbol{f}$ w.r.t. λ; `ret(i)` corresponds to $\frac{\partial f_i}{\partial \lambda}$;
- `'pathfollowing'`: struct, containing the fields:
  - `activate`: boolean, `1` if pathfollowing problem, `0` otherwise;
  - `pathdata`: handle to the function `PathCharData`, computing the characteristic value of the solution function that is being followed, for examples see Section 3.7 or [4, Sec. 1.2.2];
  - `startat`: either `'start'` or the name of a file contained in the folder specified under `dir`;
  - `start`: first value of the pathfollowing parameter taken during the run, this is only considered if `startat` is set to `'start'`;
  - `steplength`: first value of the step-length taken during the run, this is only considered if `startat` is set to `'start'`;
  - `max_pred_length`: (*can be omitted*) maximal length of the predictor step;
  - `pit_stop`: (*can be omitted*) a row vector containing 2 entries, a first value when reached by the pathfollowing parameter, the user will be prompted what to do next, and a second value when reached by the value of the function `PathCharData` during the run, the user will be also prompted what to do next;
  - `require_exact`: (*can be omitted*) a row vector of values when reached by the pathfollowing variable, the solution of the BVP at this specific pathfollowing parameter value shall be computed and saved in the MATLAB-cell variable `speicher_exact`, which is one of the 3 outputs of the function `pathfollowing`;
  - `dir`: the folder in which the data is saved or from where it may be loaded, and
  - `name`: the name under which the path's data consisting of the two cell-variables `speicher` and `speicher_exact` is saved;
  - `counter`: (*can be omitted*) the number of pathfollowing steps that are executed before the user is prompted what to do next. This can also be set to a very high number, then the code will run until a value from `pit_stop` is reached. This option is only recommended for already tested problems, since some behaviour may occur along the way otherwise, that are undesired;

  - `only_counter`: (*can be omitted*) this is a boolean value used with `counter`. If it is set to `1`, then the number of steps given in `counter` will be performed and then the path automatically saved without any further user input. If it is set to `0`, nothing will happen;
  - `only_exact`: (*can be omitted*) this is used with `require_exact` and `startat`. On the base of a previously computed path, the approximations to the solutions of the equations with the pathfollowing parameter being equal to the values in the vector `require_exact` are computed and saved.

- `'path_jac'`: $\frac{\partial \boldsymbol{f}}{\partial \lambda^*}$ The Jacobian of $\boldsymbol{f}$ w.r.t. λ^*; `ret(i)` corresponds to $\frac{\partial f_i}{\partial \lambda^*}$;
- `'path_dBV'`: $\frac{\partial \boldsymbol{g}}{\partial \lambda^*}$ The Jacobian of $\boldsymbol{g}$ w.r.t. λ^*; `ret(i)` corresponds to $\frac{\partial g_i}{\partial \lambda^*}$;

Once all the required information has been entered in `problem1.m`, we can now proceed to adjust the solver settings.

2.3.2 Solver settings

To adjust the solver settings, we use the file `default_settings.m`, which provides a template for this task. The following settings can be manipulated:

- `'mesh'`: $\boldsymbol{\tau}$ Row vector of the mesh on $[0, 1]$ on which the problem is solved;
- `'collMethod'`: Determines which collocation points are used. Currently, `'gauss'` (Gauss-Legendre points), `'lobatto'` (Lobatto points), `'uniform'` (uniformly distributed points) and `'user'` (user-specified points) are available;
- `'collPoints'`: If in `'collMethod' 'user'` is chosen, then a row vector of the points on $[0, 1]$ is specified here, or else a natural number corresponding to the number of points is specified here (i.e. m);
- `'meshAdaptation'`: `1`, if successive adaptation of the mesh should be preformed, until the tolerances (specified in `'absTolMeshAdaptation'` and `'relTolMeshAdaptation'`) are satisfied. This implies the use of error estimation, cf. Section 2.5. `0` otherwise;
- `'errorEstimate'`: `1`, if the error should be estimated, cf. Section 2.5, `0` otherwise;
- `'absTolSolver'` and
- `'relTolSolver'`: specify the tolerances used in the non-linear solver. Further information can be found in [2];
- `'absTolMeshAdaptation'` and
- `'relTolMeshAdaptation'`: specify the tolerances that are used in the mesh adaptation procedure;
- `'minInitialMesh'`: specifies the minimal number of points in the initial profile. If the user declares a profile with fewer points, the missing points are inserted and the values are computed by interpolation.

Nine further settings below these are provided for users willing to delve deeper into the code base of the non-linear solver. Since this may not be of interest for the casual user of `bvpsuite2.0`, at this point we refer to [2].
In the case of a pathfollowing problem, some more settings can be adjusted, namely

- `'thetaMax'`: value smaller than 0.25, which determines how quickly the step-length gets adjusted;
- `'maxCorrSteps'`: after the prediction of the next step-length, the new steplength may undergo maximally this number of corrections, i.e. multiplying the step-length by $\frac{1}{\sqrt{2}}$;
- `'maxSteplengthGrowth'`: upper bound for the factor by which the step-length can grow from one step to the next. If either of the following occurs

  1. the trust region method is called, which is turned off during pathfollowing,
  2. a new number of mesh points that is too high is suggested,
  3. the angle between the current step and the next tangent was too big,
  4. the corrector step was too long compared to the corrector step in the previous step,
  5. the corrector step was too long compared to the predictor step,

  then the number in `'maxSteplengthGrowth'` is divided by $\sqrt{2}$ to the power the number of times the step-length was halved in the previous step;
- `'angleMin'`: minimal cosine of the angle between current step and the tangent in the following step. If below this bound, then the step-length gets halved and the step is recomputed; this can be turned off by setting it to `-1`;
- `'meshFactorMax'`: controls by which factor the number of points in the mesh can be augmented during a single pathfollowing step. If a higher number of mesh points is suggested, then the step-length gets halved and the step is recomputed;
- `'PredLengthFactor'`: Factor which controls the length of the corrector step compared to the predictor step and the corrector step. If this number times the length of the corrector step is higher than the length of the predictor step, then the step-length gets halved and the step is recomputed; this can be turned off by setting it to `0`;
- `'CorrLengthGrowth'`: Factor which controls the maximal allowed growth of the corrector step compared to corrector step of the previous step. If this number times the length of the previous corrector step is higher than the length of the current corrector step, then the step-length gets halved and the step is recomputed; this can be turned off by setting it to `Inf`.

These are the solver settings that can be adjusted. Next, we will have a short look at the two remaining arguments of the function **bvpsuite2**.

2.3.3 Initial profile

The initial profile can be transmitted through the function call of `bvpsuite2.m` directly as a struct object. This can be handled in the following manner:

```
x_init=[1,2,3]; % Row vector of length >=2 of the initial mesh points
y_init=[1,2,3;1,2,3;1,2,3]; % n*t-matrix of the initial values for each of
    the n (here, n=3) solution components at each mesh point
s_init=struct('initialMesh',x_init,'initialValues',y_init);
[x,y,s] = bvpsuite2( 'template_bvp', 'default_settings', s_init );
```

The initial mesh and values provided in the function call will be taken instead of the initial mesh and values specified in the problem definition at 'initProfile', as described in Section 2.3.1.

2.3.4 Pathfollowing struct

In the case of a pathfollowing problem, the pathfollowing related settings in the problem definition file from 'pathfollowing' and the seven pathfollowing related settings in the solver settings file can be transmitted directly in the function call of bvpsuite2 and will replace the specific values specified in the problem definition and solver settings respectively. In this struct transmitted in the function argument, it is possible to transmit only a small number of settings, all need not be given.
For instance:

```
pathfoll.pit_stop=[1,0]; % pause, when the pathfollowing parameter reaches
    1 or @pathchardata reaches 0
pathfoll.counter=50; % make 50 pathfollowing steps before pause
pathfoll.thetaMax=0.05; % set the step-length to adjust slowly
pathfoll.PredLengthFactor=2; % limit the length of the corrector step to be
     at most half as long as the predictor step
[x,y,s] = bvpsuite2( 'template_bvp', 'default_settings', [], pathfoll );
```

An initial profile may also be defined and transmitted in the function call here.

2.4 Output

The output of the function bvpsuite2 consists of three quantities, namely x, y, and s.

x is a row vector in which the mesh points are saved. The first and last values in the vector are dependant on the input 'interval', as described in Section 2.3.1. The in-between values and the length of the vector is dependent on the parameters specified in 'mesh' and 'meshAdaptation', as explained in Section 2.3.2. If 'meshAdaptation' is set to 0, then the output x will be the mesh as specified in 'interval' and 'mesh'. Otherwise, the mesh adaptation algorithm will transform and if need be, add more mesh points, until the tolerances are satisfied, cf. Section 2.5.

y saves the solutions of the collocation method in the mesh points. The output **y** is then a matrix where the number of rows corresponds to the number of solution components, i.e. 'n' from Section 2.3.1, and the number of columns is the number of mesh points, i.e. the length of the output vector x.

s is a MATLAB-struct element, in which a few computational results are saved. These are the following:

- x1: Row vector of the mesh points; the same as **x**;
- valx1: $\boldsymbol{n}$-row matrix of the solution values at the mesh points; the same as **y**;

- `x1tau`: Row vector of the mesh and the collocation grid points in-between each two adjacent mesh points;
- `valx1tau`: $\boldsymbol{n}$-row matrix of the solution values at the mesh and collocation grid points;
- `parameters`: Row vector of the parameter values;
- `coeff`: $\boldsymbol{n}$-column matrix of the coefficients of the approximating polynomials in the Runge-Kutta basis, see [4, Sec. 1.2.1] for further details;
- `errest`: $\boldsymbol{n}$-row matrix of the automatically generated error estimation at the mesh and collocation grid points;
- `lambda`: (*only for eigenvalue problem*) the value of the eigenvalue corresponding to the approximation of the eigenfunction;
- `lambda_p`: (*only for pathfollowing problem*) value of λ^* for this solution;
- `predictor`: (*only for pathfollowing problem*) some pathfollowing values specific to this step.

At this point, we explain briefly how to evaluate the approximate solution at other points in the interval. For this task, the function `coeffToValues` can be used. Assuming we want to get the value of the approximation of the solution of the problem (1) at 0, 0.25, 0.5, 0.75 and 1. Then the following lines can be executed:

```
ordnung = feval_problem( 'template_bvp' , 'orders' ); % highest order of
    the components in the equations
rho = getStandardCollocationPoints( feval('default_settings','collMethod')
    ,  feval('default_settings','collPoints') ); % collocation points in
    [0,1]
xx = 0:0.25:1 ; % values at which we want to evaluate the piecewise
    polynomial function approximating the solution
deriv = 0 ; % at which derivative we evaluate, 0 is the function, 1 the
    first derivative, and so on
yy = coeffToValues( s.coeff , s.x1 , ordnung , rho , xx , deriv );
fprintf(' xx: %s\n yy: %s\n',num2str(xx),num2str(yy))
```

The function `coeffToValues` is also used frequently in the definition of `PathCharData`, when dealing with pathfollowing problems, in order to follow the evolution of the value of the derivative of a certain component at a certain point in the interval. If, for instance, we are interested in the evolution of $z_5^{(4)}(-3)$, this would then be realized as

```
function ret = PathCharData(x1,coeff,ordnung,rho)
help=coeffToValues( coeff, x1,ordnung,rho,-3,4);
ret = help(5);
end
```

2.5 Error estimation and mesh adaptation

The error estimation strategy will provide an asymptotically exact difference between the solution of the BVP and its approximation computed by `bvpsuite2.0`. The algorithm is based on refining the mesh by halving the intervals between mesh points.
The mesh adaptation algorithm repeatedly

1. rearranges the mesh points such that the approximation error is evenly distributed over the whole interval, or
2. adds more mesh points to improve the overall error,

and performs one of these two actions at a time, until the computed error estimate satisfies the tolerances chosen for the mesh adaptation in the problem settings file.
Both features are almost completely unchanged from `bvpsuite1.1`. Only the data structure had to be adapted. For the theoretical background we refer to [1].
The error estimation and mesh adaptation are invoked by setting the respective properties in the settings file as discussed in Section 2.3.2.
The mesh adaptation algorithm uses the results of the error estimation and performs until the tolerances from the solver settings file are satisfied. Consequently, when the mesh adaptation is used, the error estimate will be computed, regardless of the setting in `'errorEstimate'`, cf. Section 2.3.2. In contrast, the error estimate can be computed without mesh adaptation enabled. When none of the two are enabled, then `bvpsuite2.0` computes the solution values on the fixed mesh provided by the user input.

3 Specific problem definitions and examples

Some computation examples are provided within the package, in order to help the user get familiar with `bvpsuite2.0`. The example files are provided in the directory `Examples`. To call the files directly from within the directory, execute the `addpath`-function:

```
addpath( 'full path to the directory "Examples"' )
```

In the directory the following files are included:

	Problem definition for
`bvps2_lin_probdef.m`	Linear problem.
`bvps2_nonlin_probdef.m`	Non-linear problem.
`bvps2_pardep_probdef.m`	Parameter-dependent problem.
`bvps2_evp_probdef.m`	eigenvalue problem.
`bvps2_dae_probdef.m`	Differential algebraic problem.
`bvps2_semiinf_probdef.m`	Problem posed on a semi-infinite interval.
`bvps2_pathfoll_probdef.m`	Pathfollowing problem.
`bvps2_bcint_probdef.m`	Boundary conditions inside interval.

	Solver settings for
`bvps2_lin_settings.m`	Linear problem with mesh adaptation enabled.
`bvps2_lin_settings_woMA.m`	Linear problem with mesh adaptation disabled.
`bvps2_nonlin_settings.m`	Non-linear problem.
`bvps2_pardep_settings.m`	Parameter-dependent problem.
`bvps2_evp_settings.m`	eigenvalue problem.
`bvps2_dae_settings.m`	Differential algebraic problem.
`bvps2_semiinf_settings.m`	Problem posed on a semi-infinite interval.
`bvps2_pathfoll_settings.m`	Pathfollowing problem.
`bvps2_bcint_settings.m`	Boundary conditions inside interval.

We briefly present the following examples that `bvpsuite2.0` computes efficiently. The case of the problem with boundary conditions imposed inside of the problem interval is treated in Appendix A.

3.1 Linear problem

Remark: In the problem definition, `'linear'` can be set to `1`, `'initProfile'` is not relied upon and can be set to any dummy vectors.
Example: Taken from [3, Sec. 2.3].

Consider the linear, singularly perturbed BVP, with $\varepsilon = 10^{-4}$,

$$\varepsilon z_1''(t) + z_1'(t) - (1+\varepsilon) z_1(t) = 0, \quad t \in [-1, 1], \tag{4a}$$

$$z_1(-1) = 1 + \mathrm{e}^{-2}, \quad z_1(1) = 1 + \mathrm{e}^{\frac{-2(1+\varepsilon)}{\varepsilon}}. \tag{4b}$$

The file `bvps2_lin_probdef.m` is the first input argument of the **bvpsuite2** function.It contains the problem definition for this BVP. As in Section 2.3.1, we have

$$\begin{aligned}
&\boldsymbol{n} = 1, \quad \boldsymbol{l} = [2], \quad \boldsymbol{f} = \varepsilon z_1^{(2)}(t) + z_1^{(1)} - (1+\varepsilon) z_1^{(0)}(t), \\
&\frac{\partial f}{\partial z_1^{(0)}(t)} = -(1+\varepsilon), \quad \frac{\partial f}{\partial z_1^{(1)}(t)} = 1, \quad \frac{\partial f}{\partial z_1^{(2)}(t)} = \varepsilon, \\
&\boldsymbol{g} = \begin{pmatrix} z_1^{(0)}(a) - \left(1 + \mathrm{e}^{-2}\right) \\ z_1^{(0)}(b) - \left(1 + \mathrm{e}^{\frac{-2(1+\varepsilon)}{\varepsilon}}\right) \end{pmatrix}, \\
&\frac{\partial g_1}{\partial z_1^{(0)}(a)} = 1, \quad \frac{\partial g_1}{\partial z_1^{(0)}(b)} = 0, \quad \frac{\partial g_2}{\partial z_1^{(0)}(a)} = 0, \quad \frac{\partial g_2}{\partial z_1^{(0)}(b)} = 1.
\end{aligned}$$

The solver settings are specified in `bvps2_lin_settings.m`. We chose to approximate the solution to this BVP on a starting mesh with 201 equidistant mesh points and Gauss collocation with 4 collocation points in-between each two mesh points and with mesh adaptation enabled. We set the tolerances we want to achieve to 10^{-9} for the absolute and relative tolerance of the mesh adaptation. The tolerances on the non-linear solver do not need to be set, since this is a linear problem.
The following code lines can be executed to compute this example and plot the approximate solution and error estimate.

```
[x,y,s] = bvpsuite2( 'bvps2_lin_probdef', 'bvps2_lin_settings' );
plot( x, y ); % approximate solution plot
figure; semilogy( s.x1tau, abs(s.errest) ) % plot of the error estimate
```

After some adaptations of the grid, the approximate solution and error estimate are calculated and displayed in Figure 1.
Hint: In many examples, the mesh adaptation is a very useful tool. In this first example, to achieve the tolerances we specified above without mesh adaptation, we would need many more mesh points in our starting mesh, due to the solution functions behaviour near $t = -1$. For comparison, the same problem has been calculated on an equidistant mesh with the same starting mesh with 201 mesh points. The results are displayed in Figure 2. The error on the equidistant mesh is not even close to the error on the adapted grid.

3.2 Non-linear problem

Remark: `'linear'` must be set to `0` and an initial profile must be provided under the case `'initProfile'`, see Section 2.3.1 for details.

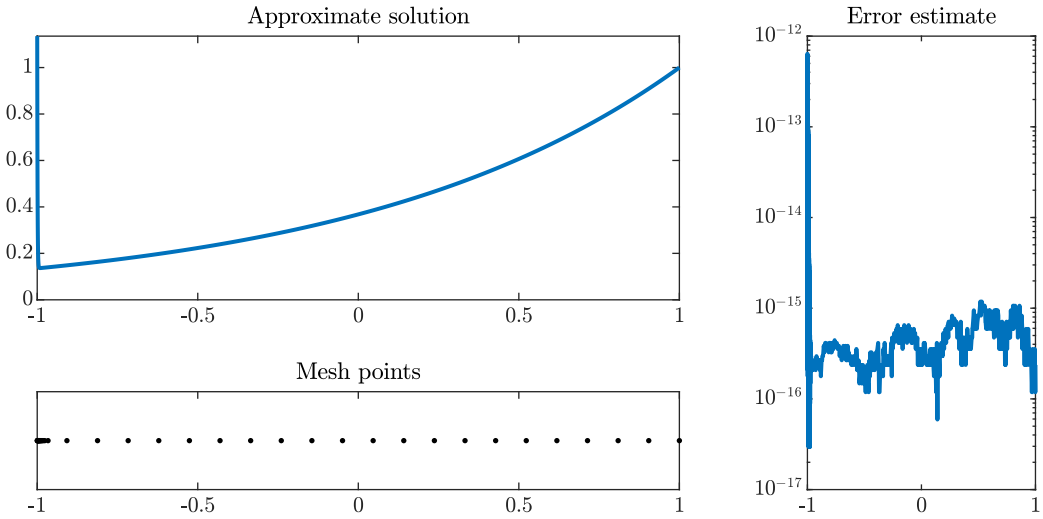

Figure 1: Linear problem (4) on the adapted mesh with $N = 201$ grid points: Plot of the approximate solution, the grid point distribution (every fifth point is displayed), and the error estimate.

Example: The following example was taken from [5].
Consider the non-linear singular BVP, where $\nu = \frac{1}{3}$, $\mu = 9$ and $\gamma = 1000$,

$$z_1''(t) + \frac{3}{t} z_1'(t) = -\mu^2 z_2(t) - 2\gamma + z_1(t) z_2(t), \quad t \in (0, 1], \tag{5a}$$

$$z_2''(t) + \frac{3}{t} z_2'(t) = \mu^2 z_1(t) - \frac{1}{2} z_1^2(t), \tag{5b}$$

$$z_1'(0) = 0, \qquad z_2'(0) = 0, \tag{5c}$$

$$z_1(1) = 0, \qquad z_2'(1) + (1 - \nu) z_2(1) = 0, \tag{5d}$$

In the file `bvps2_nonlin_probdef.m`, we define our problem with the notation intro-

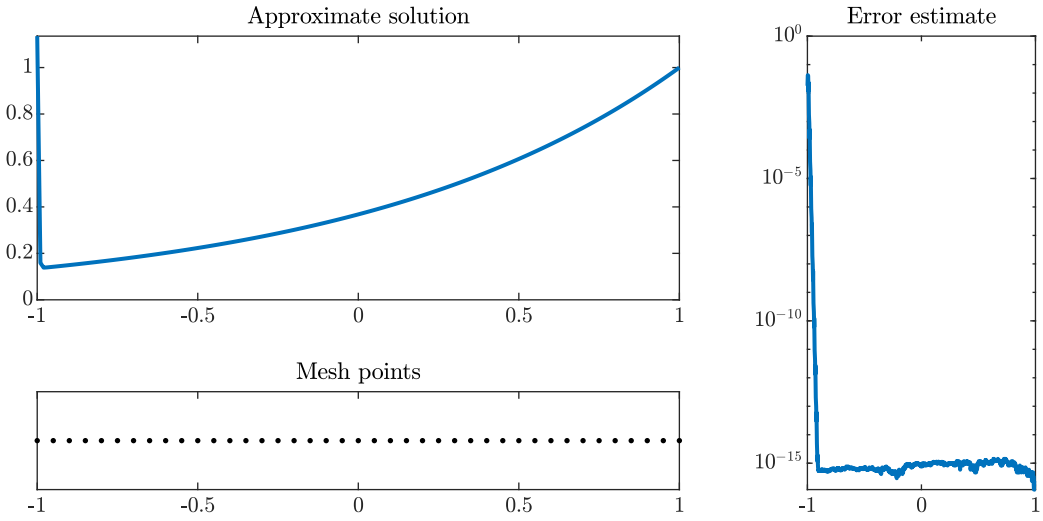

Figure 2: Linear problem (4) on an equidistant mesh with $N = 201$ grid points: Plot of the approximate solution, the grid point distribution (every fifth point is displayed), and the error estimate.

duced in Section 2.3.1. We have

$$\boldsymbol{n} = 2, \quad \boldsymbol{l} = [2,2], \quad \boldsymbol{f} := \begin{pmatrix} z_1^{(2)}(t) + \frac{3}{t} z_1^{(1)}(t) + \mu^2 z_2^{(0)}(t) + 2\gamma - z_1^{(0)}(t) z_2^{(0)}(t) \\ z_2^{(2)}(t) + \frac{3}{t} z_2^{(1)}(t) - \mu^2 z_1^{(0)}(t) + \frac{1}{2}\left(z_1^{(0)}(t)\right)^2 \end{pmatrix},$$

$$\frac{\partial f_1}{\partial z_1^{(0)}(t)} = -z_2^{(0)}(t), \quad \frac{\partial f_1}{\partial z_1^{(1)}(t)} = \frac{3}{t}, \quad \frac{\partial f_1}{\partial z_1^{(2)}(t)} = 1,$$

$$\frac{\partial f_1}{\partial z_2^{(0)}(t)} = \mu^2 - z_1^{(0)}(t), \quad \frac{\partial f_1}{\partial z_2^{(1)}(t)} = 0, \quad \frac{\partial f_1}{\partial z_2^{(2)}(t)} = 0,$$

$$\frac{\partial f_2}{\partial z_1^{(0)}(t)} = -\mu^2 + z_1^{(0)}(t), \quad \frac{\partial f_2}{\partial z_1^{(1)}(t)} = 0, \quad \frac{\partial f_2}{\partial z_1^{(2)}(t)} = 0,$$

$$\frac{\partial f_2}{\partial z_2^{(0)}(t)} = 0, \quad \frac{\partial f_2}{\partial z_2^{(1)}(t)} = \frac{3}{t}, \quad \frac{\partial f_2}{\partial z_2^{(2)}(t)} = 1,$$

$$\boldsymbol{g} := \begin{pmatrix} z_1^{(1)}(a) \\ z_2^{(1)}(a) \\ z_1^{(0)}(b) \\ z_2^{(1)}(b) + (1-\nu) z_2^{(0)}(b) \end{pmatrix},$$

$$\frac{\partial g_1}{\partial z_1^{(1)}(a)} = 1, \quad \frac{\partial g_2}{\partial z_2^{(1)}(a)} = 1, \quad \frac{\partial g_3}{\partial z_1^{(0)}(b)} = 1, \quad \frac{\partial g_4}{\partial z_2^{(0)}(b)} = 1 - \nu, \quad \frac{\partial g_4}{\partial z_2^{(1)}(b)} = 1.$$

As mentioned before, the non-linear solver needs a starting profile for each of the unknown functions of the BVP, i.e. for z_1 and z_2 in this case. In this example it suffices to set the initial approximation profile to be the constant function 1 for both unknown functions.

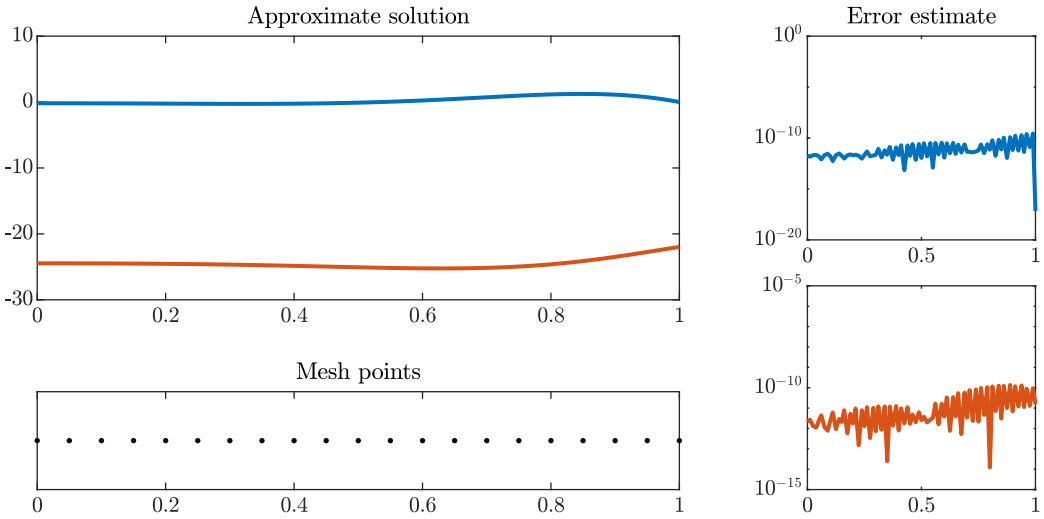

Figure 3: Nonlinear problem (5) on the mesh with $N = 101$ grid points: Plot of the approximate solution, the grid point distribution (every fifth point is displayed), and the error estimate.

Note that the division by $t = 0$ can lead to `Inf` or `NaN` values in MATLAB. This will not happen during the solver iteration for this problem, since in the above equations the singularity occurs at one of the two interval boundaries. The equations are only evaluated at the collocation points in-between the points of the discrete mesh, which starts at 0 and ends at 1. Thus, the problem can be entered as written above in the problem definition file `bvps2_nonlin_probdef.m`. If the singularity in a problem is of a higher order α, this is still not a problem, but sometimes for convergence purposes, it might be advantageous to multiply the equation by t^α.
The solver settings are specified in `bvps2_nonlin_settings.m`. We chose to find the approximation to the solution of this BVP using a starting mesh with 101 equidistant mesh points and Gauss collocation with 3 collocation points in-between each two mesh points. The relative and absolute tolerances for the non-linear solver were set to 10^{-12} and to 10^{-9} for the absolute and relative tolerance of the mesh adaptation.
The following code lines can be executed to compute this example and plot the approximate solution and error estimate.

```
[x,y,s] = bvpsuite2( 'bvps2_nonlin_probdef', 'bvps2_nonlin_settings' );
plot( x, y ); % approximate solution plot
figure; semilogy( s.x1tau, abs(s.errest) ) % plot of the error estimate
```

The final mesh still contains 101 mesh points and the points were not relocated. The approximate solution and error estimate are displayed in Figure 3.
Hint: The quality of the initial profile is essential for the convergence of unsmooth problems. For some complex problems, it may be necessary to acquire a relatively accurate initial profile by other means and start the computation with `bvpsuite2.0` from

there. For instance, in [6], the non-linear equations were first linearised, a solution for the linear problem was computed by `bvpsuite2.0` and then used as an initial profile for the non-linear BVP. It should be mentioned, that the linearized problem is in this manual the example used in Appendix A to discuss the numerical solution with `bvpsuite2.0` of a problem with boundary conditions imposed inside of the interval on which the equations hold.

3.3 Parameter-dependent problem

Remark: The boundary conditions have to be augmented by one condition for each parameter, such that the parameter-dependent BVP has a unique solution. `'parameters'`, `'c'`, `'dP'`, `'dP_BV'` may need some adjustment, see Section 2.3.1 for details.
Example: The following example was taken from [3, Sec. 2.4].
Consider the BVP

$$z_1'(t) = -\frac{p}{1-p}\frac{\sqrt{t}-z_1(t)}{t-z_1(t)}, \tag{6a}$$

$$z_2'(t) = z_1(t), \quad \text{for} \quad t \in [0,1], \tag{6b}$$

$$z_1(0) = 0, \quad z_2(0) = 0, \quad z_2(1) = p. \tag{6c}$$

In the file `bvps2_pardep_probdef.m`, we define our problem with the notation introduced in Section 2.3.1. We have

$$\boldsymbol{n} = 2, \quad \boldsymbol{l} = [1,1], \quad \boldsymbol{s} = 1, \quad \boldsymbol{f} := \begin{pmatrix} z_1^{(1)}(t) + \frac{p_1}{1-p_1}\frac{\sqrt{t}-z_1^{(0)}(t)}{t-z_1^{(0)}(t)} \\ z_2^{(1)}(t) - z_1^{(0)}(t) \end{pmatrix},$$

$$\frac{\partial f_1}{\partial z_1^{(0)}(t)} = \frac{p_1}{1-p_1}\frac{\sqrt{t}-k(t)}{(t-z_1^{(0)}(t))^2}, \quad \frac{\partial f_1}{\partial z_1^{(1)}(t)} = 1, \quad \frac{\partial f_2}{\partial z_1^{(0)}(t)} = -1, \quad \frac{\partial f_2}{\partial z_2^{(1)}(t)} = 1,$$

$$\boldsymbol{g} := \begin{pmatrix} z_1^{(0)}(a) \\ z_2^{(0)}(a) \\ z_2^{(0)}(b) - p_1 \end{pmatrix}, \quad \frac{\partial g_1}{\partial z_1^{(0)}(a)} = 1, \quad \frac{\partial g_2}{\partial z_2^{(0)}(a)} = 1, \quad \frac{\partial g_3}{\partial z_2^{(0)}(b)} = 1,$$

$$\frac{\partial f_1}{\partial p_1} = (1-p_1)^{-2}, \quad \frac{\partial g_3}{\partial p_1} = -1.$$

As initial profile the constant function 1 was chosen and the value 1 for the parameter, i.e.

```
ret.initialMesh = linspace(0,1,6);
ret.initialValues = ones(2,6);
ret.parameters = 1;
```

Note that in this example, if p is initialized with 1 or whenever it reaches the value 1, a division by 0 would occur and the code would return an error. Since the initial value 1 was chosen for the parameter, this problem is remedied by multiplying the first equation by $(1-p)$ in the problem definition and thus there would not be any division by 0

occurring during the procedure. This leads to the code in `bvps2_pardep_probdef.m`. The solver settings can be referred to in `bvps2_pardep_settings.m`. We chose to compute an approximation to the problem with a starting mesh with 101 equidistant mesh points and Gauss collocation with 3 collocation points in-between each two mesh points. The tolerances were set to 10^{-6} for the absolute and relative tolerance of the non-linear solver and to 10^{-4} for the absolute and relative tolerance of the mesh adaptation.
The following code lines can be executed to compute this example and plot the approximate solution and error estimate.

```
[x,y,s] = bvpsuite2( 'bvps2_pardep_probdef', 'bvps2_pardep_settings' );
plot( x, y(1,:) ); % approximate solution plot
figure; semilogy( s.x1tau, abs(s.errest(1,:)) ); % plot of the error
    estimate
disp(['Value of p_1: ' , num2str(s.parameters)])
```

The computation finishes after some mesh adaptation with a final mesh with 101 points, where the mesh points were slightly displaced, but none were added. The approximate solution, error estimate and the mesh points distribution plots are displayed in Figure 4.

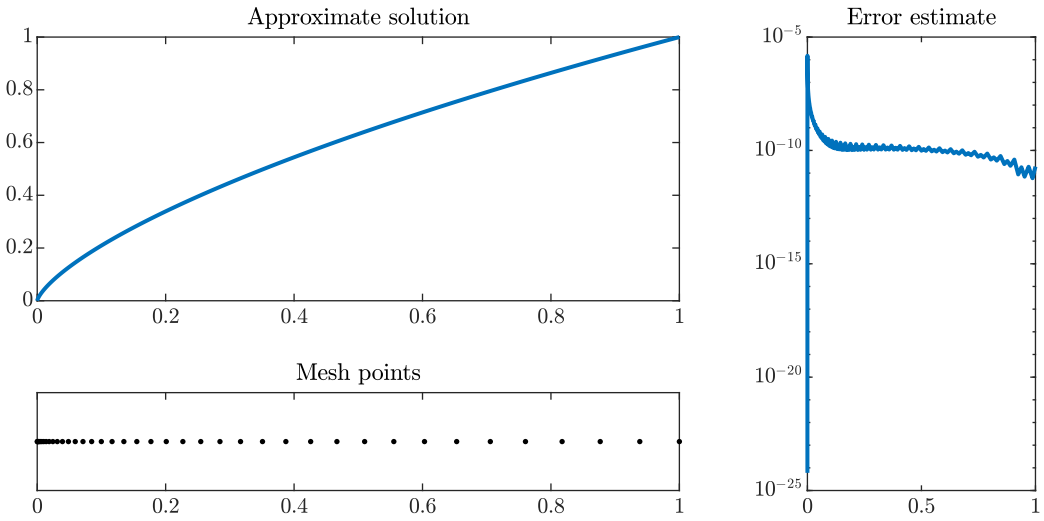

Figure 4: Parameter-dependent problem (6) on the adapted mesh with $N = 101$ grid points: Plot of the approximate solution, the grid point distribution (every fifth point is displayed), and the error estimate.

3.4 Eigenvalue problem

Remark: The eigenvalue λ is treated as an unknown parameter. When the flag `'EVP'` is set to `1`, `bvpsuite2.0` transforms the underlying EVP into a parameter-dependent non-linear BVP. Then one can use the variable λ to describe the problem. Note that

additionally an initial profile `'initProfile'` and the output parameter `'dLambda'` has to be set, which describes the derivative of $\boldsymbol{f}$ with respect to the eigenvalue, as explained in 2.3.1. For more information on the handling of eigenvalue problems by `bvpsuite2.0`, see [3, Chap. 4].

Example: The following problem was taken from [3, Sec. 4.2].
Consider the EVP

$$-z_1''(t) + \frac{c}{t^2} z_1(t) = \lambda z_1(t), \quad t \in (0, \pi), \quad c > 0, \tag{7a}$$

$$z_1(0) = z_1(\pi) = 0. \tag{7b}$$

In the file `bvps2_evp_probdef.m`, we define our problem with the notation introduced in Section 2.3.1. We have

$$\boldsymbol{n} = 1, \quad \boldsymbol{l} = [2], \quad \boldsymbol{f} := -z_1^{(2)}(t) + \frac{c}{t^2} z_1^{(0)}(t) - \lambda z_1^{(0)}(t),$$

$$\frac{\partial f}{\partial z_1^{(0)}(t)} = \frac{c}{t^2} - \lambda, \quad \frac{\partial f}{\partial z_1^{(1)}(t)} = 0, \quad \frac{\partial f}{\partial z_1^{(2)}(t)} = -1,$$

$$\boldsymbol{g} := \begin{pmatrix} z_1^{(0)}(a) \\ z_1^{(0)}(b) \end{pmatrix}, \quad \frac{\partial g_1}{\partial z_1^{(0)}(a)} = 1, \quad \frac{\partial g_2}{\partial z_1^{(0)}(b)} = 1,$$

$$\frac{\partial f}{\partial \lambda} = -z_1^{(0)}(t).$$

Since we are dealing with an eigenvalue problem, the linear problem (7) is transformed to a non-linear problem, as mentioned in the remark above. Thus, an initial profile has to be provided. As an initial profile in `'initProfile'` we chose

```
ret.initialMesh = [0 pi];
ret.initialValues = [1 1];
ret.lambda = 1;
```

Note that also an initial value for the eigenvalue λ has to be specified.
The solver settings are specified in `bvps2_evp_settings.m`. We chose to approximate the solution to this EVP using a starting mesh with 51 equidistant mesh points and Gauss collocation with 3 collocation points in-between each two mesh points. The tolerances are set to 10^{-12} for the absolute and relative tolerance of the non-linear solver and to 10^{-9} for the absolute and relative tolerance of the mesh adaptation.
The following code lines can be executed to compute this example and plot the approximate solution and error estimate:

```
[x,y,s] = bvpsuite2( 'bvps2_evp_probdef' , 'bvps2_evp_settings' );
plot(x, y); % approximate solution plot
figure; semilogy( s.x1tau, abs(s.errest(1,:)) ); % plot of the error
    estimate
```

Note that the error estimate matrix in `s.errest` has 2 rows at the end of the computation, albeit the system (7) only has a single unknown function. This is due to the

equation line that was added automatically by `bvpsuite2.0`, when dealing with eigenvalue problems, as explained in the remark above. In the struct `s`, the fields `valx1` and `valx1tau`, become matrices with 2 rows.
The mesh is adapted during the computation to a final mesh containing 227 points. The result is displayed in Figure 5.

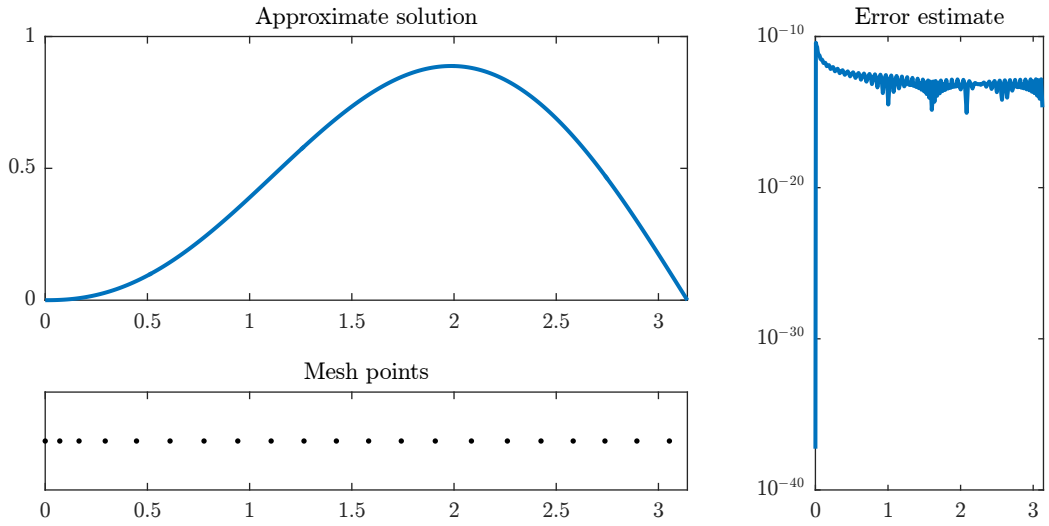

Figure 5: Eigenvalue problem (7) on the adapted mesh with $N = 227$ mesh points: Plot of the approximate solution, the mesh point distribution (every fifth point is displayed), and the error estimate.

There is an indefinite number of pairs of eigenfunctions and eigenvalues, and finding the desired pair of eigenfunction and eigenvalue, based on the initial guess, may not be straightforward. Therefore, for linear EVPs, the function `computeEVPStart` has been developed, which can provide appropriate initial guesses for the sought eigenvalue-eigenfunction pair. In fact, the function `computeEVPStart` returns a number N of initial guesses, which are accurate starting profiles for approximations of a solution pair. The problem definition, the solver settings and this number N, constitute the 3 input arguments of `computeEVPStart`. The first output argument is a cell-array of initial profiles for the eigenfunctions and the second output argument is a vector containing approximations to the eigenvalues.
Before calling the function `computeEVPStart`, we suggest to change the tolerances in `bvps2_evp_settings.m` to 10^{-9} for the absolute and relative tolerances of the nonlinear solver and 10^{-6} for the absolute and relative tolerances of the mesh adaptation. This measure is solely taken for the purpose of keeping computational time low and is not required for the execution of the following lines of code.
Choosing the number N to be 7, the following lines of code are executed:

```
N=7;
[initProfiles,initialEVs] = computeEVPStart( 'bvps2_evp_probdef' , '
```

```
    bvps2_evp_settings' , N );
initProfiles = initProfiles(1:N);
initialEVs = initialEVs(1:N);
```

`computeEVPStart` may compute more than N eigenvalue-eigenfunction pairs. Therefore only the first N are kept. For more information on this function, see [3, Sec. 4.2.1].
Now the results of this computation can be used as initial guesses for the problem (7). These initial profiles are passed to the function `bvpsuite2` as its third argument, as described in Section 2.3.3. The lines of code performing this action and plotting the results are:

```
EVs = zeros(N,1); sols = cell(1,N); legends = cell(1,N);
figure; hold on
for ii=1:N
    fprintf('\r\nSolve the EVP for the eigenvalue #%d:\n',ii);
    [x,y,sols{ii}] = bvpsuite2( 'bvps2_evp_probdef' , 'bvps2_evp_settings'
        , initProfiles{ii} );
    EVs(ii) = sols{ii}.lambda;
    legends{ii} = sprintf('\\lambda_{%d} = %4.1f',ii,sols{ii}.lambda);
    plot( x , y )
end
```

Also for any of these functions, the error estimate for each approximate solution can be plotted via

```
ii =   ; figure
plot( sols{ii}.x1tau , abs(sols{ii}.errest) )
```

The results of the computation above are displayed in Figure 6. The corresponding error estimate plots to each function are displayed in Figure 7.
The values of the initially provided eigenvalues by `computeEVPStart` were accurate for the first couple of eigenvalues, but later they are less so, as can be seen in the table below.

	Inital EV	Final EV	Initial - final EV
λ_1	2.417106	2.417106	5.9059e-08
λ_2	6.723654	6.723653	6.3428e-07
λ_3	13.027504	13.027501	3.5739e-06
λ_4	21.330745	21.330728	1.7104e-05
λ_5	31.633815	31.633736	7.8936e-05
λ_6	43.936988	43.936647	3.4138e-04
λ_7	58.240947	58.239508	1.4392e-03

During the run, the mesh was adapted significantly, as is shown in the table below.

Run for	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6	λ_7
# mesh points	51	51	51	66	74	84	93

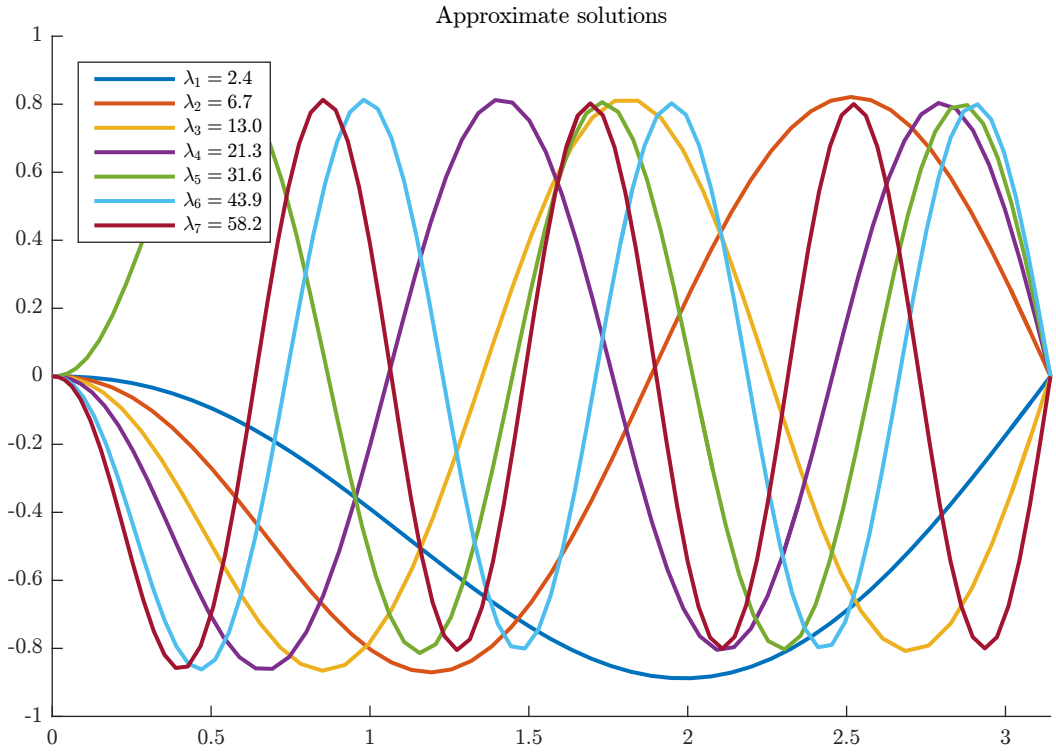

Figure 6: Eigenvalue problem (7): Approximated solutions, computed by starting at the intial profiles given by `computeEVPStart`.

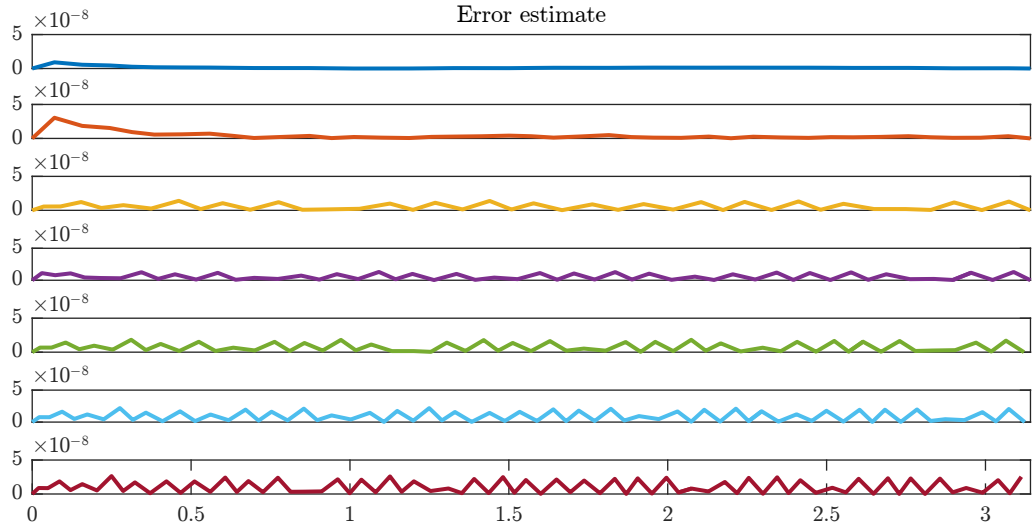

Figure 7: Eigenvalue problem (7): Error estimate for the functions from Figure 6.

The mesh was not adapted at all in the runs for λ_1 and λ_2. In the run for λ_3 the mesh points were relocated, but none were added. In each of the last 4 runs the mesh points were displaced and a few points were added.

3.5 Index-1 DAE

Remark: The solution of a system of DAEs does not require any special options, as DAEs are a special case of mixed order systems.
Example: The following example was taken from [3, Sec. 6.1].
Consider the non-linear index-1 DAE system, where $J = \frac{1}{2}$ and $\rho = 3$,

$$z_1'(t) - z_2(t)z_3(t) = 0, \tag{8a}$$
$$z_2'(t) - z_3(t) + 1 = 0, \tag{8b}$$
$$z_1(t) - \frac{J^2}{z_3(t)} - z_3(t) = 0, \quad \text{for} \quad t \in [0, 10.3]\,, \tag{8c}$$
$$z_1(0) - \frac{J^2}{\rho} - \rho = 0, \quad \text{and} \quad z_1(10.3) - \frac{J^2}{\rho} - \rho = 0, \tag{8d}$$

In the file `bvps2_dae_settings.m`, we define our problem with the notation introduced in Section 2.3.1. We have

$$\boldsymbol{n} = 3, \quad \boldsymbol{l} = [1,1,0], \quad \boldsymbol{f} := \begin{pmatrix} z_1^{(1)}(t) - z_2^{(0)}(t)z_3^{(0)}(t) \\ z_2^{(1)}(t) - z_3^{(0)}(t) + 1 \\ z_1^{(0)}(t) - \frac{1}{4z_3^{(0)}(t)} - z_3^{(0)}(t) \end{pmatrix},$$
$$\frac{\partial f_1}{\partial z_1^{(1)}(t)} = 1, \quad \frac{\partial f_1}{\partial z_2^{(0)}(t)} = -z_3^{(0)}(t), \quad \frac{\partial f_1}{\partial z_3^{(0)}(t)} = -z_2^{(0)}(t), \quad \frac{\partial f_2}{\partial z_2^{(1)}(t)} = 1,$$
$$\frac{\partial f_2}{\partial z_3^{(0)}(t)} = -1, \quad \frac{\partial f_3}{\partial z_1^{(0)}(t)} = 1, \quad \frac{\partial f_3}{\partial z_3^{(0)}(t)} = \frac{1}{4\left(z_3^{(0)}\right)^2} - 1,$$
$$\boldsymbol{g} := \begin{pmatrix} z_1^{(0)}(a) - \frac{J^2}{\rho} - \rho \\ z_1^{(0)}(b) - \frac{J^2}{\rho} - \rho \end{pmatrix}, \quad \frac{\partial g_1}{\partial z_1^{(0)}(a)} = 1, \quad \frac{\partial g_2}{\partial z_1^{(0)}(b)} = 1.$$

As initial profiles for the non-linear solver, the constant functions $\tilde{z}_1 = 1.25$, $\tilde{z}_2 = 0$ and $\tilde{z}_3 = 1$ are chosen. These satisfy the equations, but not the boundary conditions.
The solver settings are specified in `bvps2_pardep_settings.m`. The solution to this BVP was approximated using a starting mesh with 51 equidistant mesh points and a uniform collocation method with 3 collocation points in-between each two mesh points. The tolerances were set to 10^{-9} for the absolute and relative tolerance of the non-linear solver and to 10^{-6} for the absolute and relative tolerance of the mesh adaptation.
We can compute and plot our example by running

```
[x,y,s] = bvpsuite2( 'bvps2_dae_probdef', 'bvps2_dae_settings' );
plot(x, y); % approximate solution plot
figure; semilogy( s.x1tau, abs(s.errest)); % plot of the error estimate
```

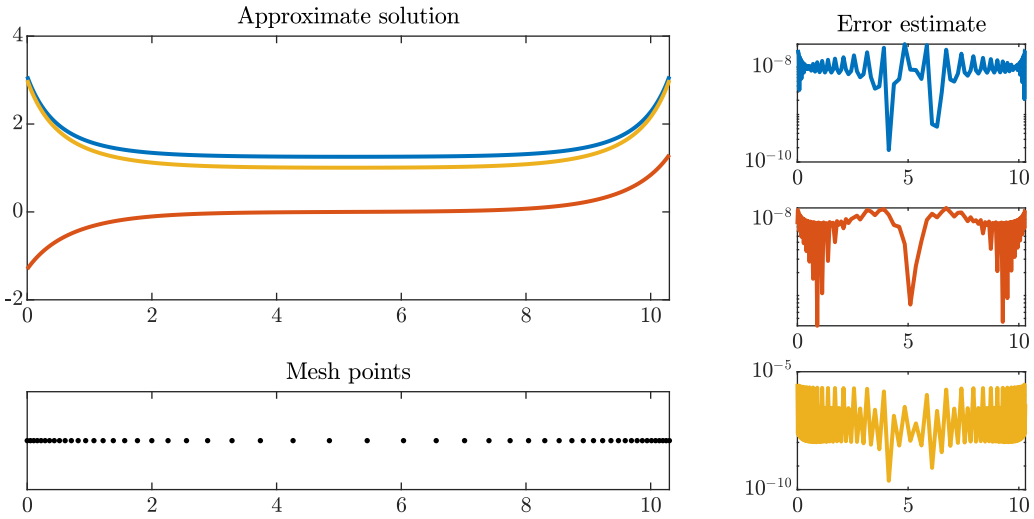

Figure 8: Index-1 DAE (8) on the adapted mesh with $N = 154$ grid points: Plot of the approximate solution, the grid point distribution (every fifth point is displayed), and the error estimate.

The mesh is adapted to a final mesh containing 154 points. The result of this computation is displayed in Figure 8.

3.6 Problem on semi-infinite interval

Remark 1: Set `'interval'` to `[a,Inf]`, where `a` is the left border.
Remark 2: To compute an approximate solution to these type of problems, `bvpsuite2.0` transforms the semi-infinite interval to a finite interval, on which the solution is then computed and afterwards transformed back to the original interval. Two cases are distinguished:

- if $a > 0$, then the standard transformation implemented in `bvpsuite2.0` is $\xi : [a, \infty) \to (0, 1], \quad t \mapsto \frac{a}{t}$, and
- if $a = 0$, then the interval $[0, \infty)$ is split up into the intervals $[0, 1]$ and $[1, \infty)$, which is then transformed by ξ to $[0, 1]$. The number of equations and solution components is doubled by this transformation. Therefore additional boundary conditions at 1 to guarantee smoothness of the approximation and its derivatives are imposed.

These measures are all performed automatically by the code as soon as the second value in `'interval'` is `Inf`. More details to this approach are available in [3, Chap. 3].
The transformation of the BVP and in the case $a = 0$, the addition of the boundary conditions, are all carried out automatically by `bvpsuite2.0`. The user solely needs to provide the problem definition and the solver settings, as demonstrated in the following example.
Example: The following example was taken from [3, Sec. 3.4].

Consider the singular BVP

$$z_1''(t) + \frac{2}{t} z_1'(t) - 4(z_1(t)+1)z_1(t)(z_1(t)-0.1) = 0, \quad t \in (0,\infty), \tag{9a}$$

$$z_1'(0) = 0, \quad z_1(\infty) = 0.1. \tag{9b}$$

In the notation of Section 2.3.1, here we have

$$\boldsymbol{n} = 1, \quad \boldsymbol{l} = [1], \quad \boldsymbol{f} := z_1^{(2)}(t) + \frac{2}{t} z_1^{(1)}(t) - 4(z_1^{(0)}(t)+1)z_1^{(0)}(t)(z_1^{(0)}(t)-0.1) = 0,$$

$$\frac{\partial f_1}{\partial z_1^{(0)}(t)} = -4z_1^{(0)}(t)(z_1^{(0)}(t)-0.1) - 4(z_1^{(0)}(t)+1)(z_1^{(0)}(t)-0.1) - 4(z_1^{(0)}(t)+1)z_1^{(0)}(t),$$

$$\frac{\partial f_1}{\partial z_1^{(1)}(t)} = \frac{2}{t}, \quad \frac{\partial f_1}{\partial z_1^{(2)}(t)} = 1, \quad \boldsymbol{g} := \begin{pmatrix} z_1^{(1)}(a) - 0 \\ z_1^{(0)}(b) - 0.1 \end{pmatrix}, \quad \frac{\partial g_1}{\partial z_1^{(1)}(a)} = 1, \quad \frac{\partial g_2}{\partial z_1^{(0)}(b)} = 1.$$

The remaining parameters of the problem definition are specified in the file `bvps2_semiinf_probdef.m`. Following [3, Sec. 3.4], the initial approximation is

```
ret.initialMesh = [ 0.0225    0.1000    0.1775    0.2000    0.2225
    0.3000    0.3775    0.4000    0.4225    0.5000    0.5775   0.6000
    0.6225    0.7000    0.7775    0.8000    0.8225    0.9000    0.9775
    1.0000    1.0231    1.1111    1.2157    1.2500    1.2862    1.4286
    1.6063    1.6667    1.7317    2.0000    2.3666    2.5000    2.6493
    3.3333    4.4936    5.0000    5.6351   10.0000 45];
ret.initialValues = [-0.3042   -0.3037   -0.3024   -0.3020   -0.3014
    -0.2991   -0.2962   -0.2952   -0.2942   -0.2902   -0.2857  -0.2842
    -0.2828   -0.2773   -0.2713   -0.2694   -0.2675   -0.2607   -0.2535
    -0.2513   -0.2490   -0.2400  -0.2286   -0.2248   -0.2207   -0.2041
    -0.1825   -0.1750   -0.1669   -0.1336   -0.0899   -0.0751   -0.0592
    0.0007    0.0593    0.0729    0.0834    0.0994 0.1];
```

The initial profile is chosen in this way, since when choosing an initial profile similar to the ones chosen in the previous examples, i.e. a constant function, the solver will converge towards the constant function $z(t) \equiv 0.1$, which is a solution to the BVP (9). So, making an educated guess for the initial profile is critical in some examples, if more than one solution exists, to find the desired one, and sometimes even to facilitate convergence of the solver.

The solver settings are in the file `bvps2_semiinf_settings.m`. The approximation of the solution to this BVP was found using a starting mesh with 51 equidistant mesh points and Gauss collocation using 5 collocation points in-between each two mesh points. The tolerances were set to 10^{-10} for the absolute and relative tolerance of the non-linear solver and to 10^{-9} for the absolute and relative tolerance of the mesh adaptation.

The following code lines can be executed to compute this example and plot the approximate solution and error estimate.

```
[x,y,s] = bvpsuite2( 'bvps2_semiinf_probdef', 'bvps2_semiinf_settings' );
```

```
plot( x, y ); % approximate solution plot
temp=size(s.errest,2);
figure; plot( s.x1tau(1,1:temp), abs(s.errest(1,:)), 'b', s.x1tau(1,temp:
    end), abs(s.errest(2,end:-1:1)), 'b') % plot of the error estimate
```

The tolerances are satisfied without any mesh adaptation. The approximate solution and error estimate plots are displayed in Figure 9.

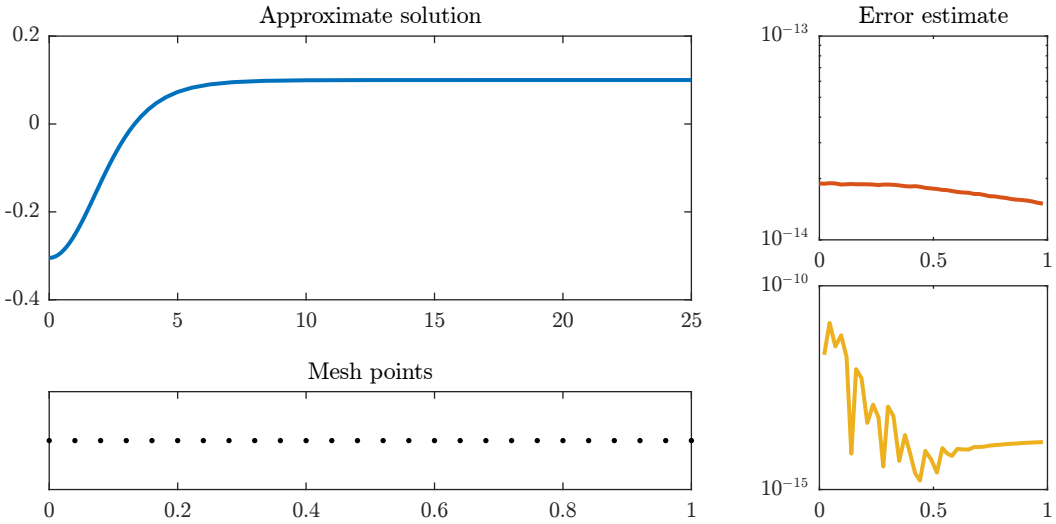

Figure 9: Semi-infinite interval problem (9): Plot of the approximate solution, the grid point distribution (every fifth point is displayed), and the error estimate.

3.7 Pathfollowing

Remark: For a pathfollowing problem, in the problem definition, in the struct under `'pathfollowing'`, the boolean value `activate` needs to be set to `1`. All the other useful fields of the struct need to be filled in. Also `'path_jac'` and `'path_dBV'` need to be filled in correctly. The seven settings specific to pathfollowing in the solver settings may be adjusted with care, depending on the problem, see Section 2.3.1 and Section 2.3.2 for details on all these options.

Example: The problem used as an example here is taken from [7].
Consider the BVP for $t \in [0, 1]$,

$$t^2 z_1^{(3)}(t) - t z_1''(t) + z_1'(t) - t^3 p - \lambda^* \left(t z_1'(t)^2 - t z_1(t) z_1''(t) + z_1(t) z_1'(t) \right) = 0, \tag{10a}$$

$$z_1(0) = z_1'(0) = 0, \quad z_1(1) = 0 \quad \text{and} \quad z_1'(1) = 1. \tag{10b}$$

In the file `bvps2_pathfoll_probdef.m`, we define our problem with the notation intro-

duced in Section 2.3.1. We have

$$\boldsymbol{n} = 1, \quad \boldsymbol{l} = [3], \quad \boldsymbol{s} = 1,$$

$$\boldsymbol{f} := t^2 z_1^{(3)}(t) - t z_1^{(2)}(t) + z_1^{(1)}(t) - t^3 p_1 - \lambda_p \left(t z_1^{(1)}(t)^2 - t z_1^{(1)}(t) z_1^{(2)}(t) + z_1^{(0)}(t) z_1^{(1)}(t) \right),$$

$$\frac{\partial f_1}{\partial z_1^{(0)}(t)} = -\lambda_p z_1^{(1)}(t), \quad \frac{\partial f_1}{\partial z_1^{(1)}(t)} = 1 - \lambda_p \left(t 2 z_1^{(1)}(t) - t z_1^{(2)}(t) + z_1^{(0)} \right),$$

$$\frac{\partial f_1}{\partial z_1^{(2)}(t)} = -t - t\lambda_p z_1^{(1)}(t), \quad \frac{\partial f_1}{\partial z_1^{(3)}(t)} = t^2,$$

$$\boldsymbol{g} := \begin{pmatrix} z_1^{(0)}(a) \\ z_1^{(1)}(a) \\ z_1^{(0)}(b) \\ z_1^{(1)}(b) - 1 \end{pmatrix}, \quad \frac{\partial g_1}{\partial z_1^{(0)}(a)} = 1, \quad \frac{\partial g_2}{\partial z_1^{(1)}(a)} = 1, \quad \frac{\partial g_3}{\partial z_1^{(0)}(b)} = 1, \quad \frac{\partial g_4}{\partial z_1^{(1)}(b)} = 1,$$

$$\frac{\partial f_1}{\partial p_1} = -t^3, \quad \frac{\partial f_1}{\partial \lambda_p} = -\left(t z_1^{(1)}(t)^2 - t z_1^{(1)}(t) z_1^{(2)}(t) + z_1^{(0)}(t) z_1^{(1)}(t) \right), \quad \frac{\partial \boldsymbol{g}}{\partial \lambda_p} = \mathbf{0}.$$

As initial profile for the unknown function, the constant function equal to 1 is chosen. For the parameter p, the initial value 8 is chosen.
In `'pathfollowing'`, the struct field `activate` is set to `1`. In the pathfollowing, we follow the evolution of the parameter p. This is implemented through the function `g`, i.e.

```
function ret = PathCharData(x1,coeff,ordnung,rho)
ret= coeff(end-1);
end
```

and the handle to this function `@PathCharData` is then set at the field `data`. Instead of following the value of a parameter, the function `coeffToValues` can be used to evaluate the piecewise polynomial approximating the solution, as is demonstrated in Section 2.4. For other examples of pathfollowing functions, see the functions `PathCharData` in [4, Sec. 1.2.2].
The field `start` is set to `0` and the field `startat` to `'start'`. Also the field `steplength` is set to `1`. This means the parameter λ_p is varied starting from 0 with initial step-length 1.
The field `require_exact` is set to `[0 8 100]`. Every time one of the values in this vector is passed, an approximation to the solution of the equations with λ_p set to this exact value is computed. These values are then also saved and can be called as shown below. In the run which is executed below, the values 0 and 8 are crossed, but not the value 100.
Also the value for the field `pit_stop` is set to `[100,-20]`. This means, that as soon as the pathfollowing variable crosses the value $\lambda_p = 100$ or if the return value of the function `PathCharData` crosses the value -20 along the path, then the function will stop at this point and prompt the user, whether to continue or stop and save the path. In the run executed below, the value -20 will be reached by `PathCharData` before the value

$\lambda_p = 100$ is reached.
The field `counter` is set to `Inf`. This means that the pathfollowing routine will run indefinitely, until the values in `pit_stop` are crossed. Then it will stop and save the function, without a prompt. So setting `counter` to `Inf` is meant to be used in a script setting, where no user interaction is desired, and the values from `pit_stop` are meant to be reached and after that the path saved and the program stopped. This is used here, because the path is already known to reach one of the values in `pit_stop` from a previous computation. For the first runs of a function, this is not a recommended setting. Alternatively, in a script, a number of steps can be executed by setting the option `counter` to this number and `only_counter` to `1`. This will make the function perform `counter`-pathfollowing steps and then save the results and stop the computation.
Finally the fields `dir` and `name` should be filled out according to where the results of the computed path should be saved and under which name. The field `dir` must be filled out by the path to the directory and it must end with a slash `/`. The field `name` is set to `'path_ex'` for this problem. A number will be added at the end of the file name after each save, so the files are not overwritten. Once the computation ends, the path is saved in the directory under the name which is given. From there the next run can be started, as is shown below.
The solver settings can be adjusted in `bvps2_pathfoll_settings.m`. We chose a starting mesh with 101 equidistant mesh points, 3 Gaussian collocation points in-between each two of these mesh points and the relative and absolute tolerances for the non-linear solver set to 10^{-6} and the relative and absolute tolerances for the mesh adaptation set to 10^{-4}. In the pathfollowing specific settings, θ_{max} was set to 10^{-2}. All the other settings assume reasonable values. For more details on those settings, see Section 2.3.2.
Then the pathfollowing run can be called by the codelet

```
[x,y,s] = bvpsuite2( 'bvps2_pathfoll_probdef', 'bvps2_pathfoll_settings' );
plot( x, y ); % plots the last solution in the path
figure; semilogy( s.x1tau,  abs(s.errest) ); % plot of the error estimate
    of the last solution in the path
disp(['Value of p_1 in last step: ' , num2str(s.parameters)])
```

During the run, the plots of the approximations of the solution at the values of λ_p equal to 0 and twice at 8, once before the turning point and once after the turning point, are displayed. Also, at the end of the run, a figure is displayed, collecting in one plot all of the information relevant to the path. This is

- on the left hand side the eight plots of some key values that were logged during the run,
- on the upper right hand side, the plot of the evolution of the function `g` and the three dimensional plot of the function's evolution under variation of the parameter λ_p, and
- on the lower right hand side, the evolution of the mesh density in each step and the evolution of the mesh points distribution.

These plots or parts of the group-plot can be suppressed and also some other options, in the lines 3–12 of the file `pathfollowing.m`.

This run finished after 13 tangent continuation steps. The resulting path of the evolution of this run can be plotted by the codelet

```
load( [ 'path to the directory /' 'path_ex_1' ] )
a=cell2mat(p_save(2,1:end-1));
b=cell2mat(p_save(3,1:end-1));
figure
plot(a,b,'b')
```

Also, the predictor and corrector steps can be shown by executing

```
hold on
c=cell2mat(p_save{1,end}.val(1,:));
d=cell2mat(p_save{1,end}.val(2,:));
for ii=1:length(c)
    plot([a(ii),c(ii)],[b(ii),d(ii)],'g')
    plot([c(ii),a(ii+1)],[d(ii),b(ii+1)],'r')
end
plot(a,b,'b-*')
```

The results of this run are displayed in Figure 10.

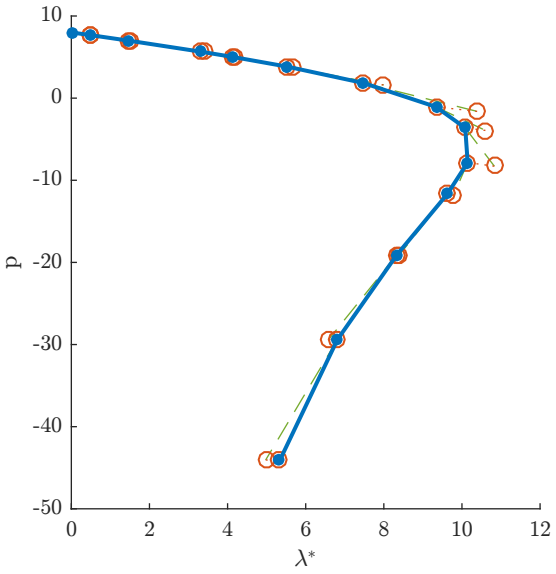

Figure 10: Pathfollowing problem (10): Evolution of the parameter p under variation of the pathfollowing parameter λ^*, until $p = -20$ is reached.

After this run, if the user wishes to go a little further, it is possible to call the just computed results by running the following lines of code:

```
pathfoll.counter = 1;
pathfoll.startat = 'path_ex_1';
pathfoll.pit_stop = [100,-60];
[x,y,s] = bvpsuite2( 'bvps2_pathfoll_probdef', 'bvps2_pathfoll_settings',
    [], pathfoll );
```

Note that here the input for the initial profile is passed as an empty vector `[]`, it is not of use here, since we are starting at the previously computed path.
After computing one step, as is set through `counter`, the user is prompted whether they want to go back a number of steps. By pressing `b`, the enter-key and afterwards a number, it is possible to go back a few steps along the path. But in this case, the path evolves reasonably, so we simply press the enter-key.
A second prompt appears, to ask whether

- to change the number of steps computed at once,
- to introduce/change the maximal allowed length of the predictor step,
- to plot the approximation in one of the computed steps,
- to stop here and save or
- to simply continue by pressing the enter-key.

We chose to continue. One more step is computed and the value -60 is reached. The first prompt is again answered with the enter-key. The second one with `s`. The path is saved under the name `path_ex_2`. The result is displayed in Figure 11.

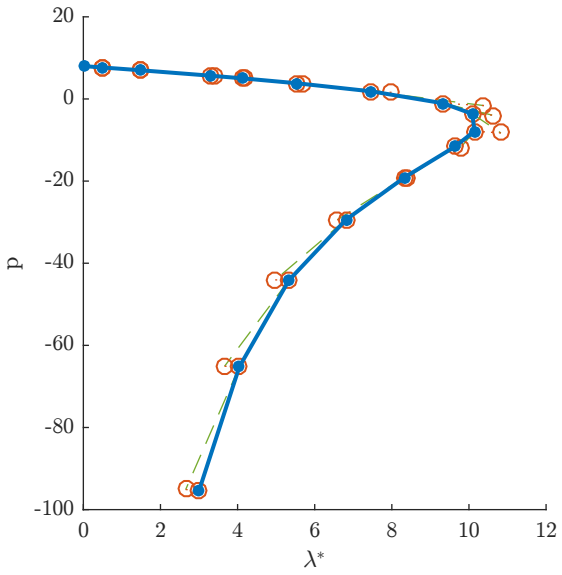

Figure 11: Pathfollowing problem (10): Evolution of the parameter p under variation of the pathfollowing parameter λ^*, until $p = -60$ is reached.

Having computed this path, we wish to see the approximate solutions for the values of the

pathfollowing parameter λ_p equal to 0, 5 and 10. By setting the field `only_exact` to `1`, the algorithm will run through the path from the start to the last point, and whenever the pathfollowing parameter crosses one of these values, the approximate solution is computed at this exact value. Afterwards, the path and the approximations at these values are saved and the program terminates by itself. The lines of code executed here are:

```
pathfoll.require_exact = [0 5 10];
pathfoll.only_exact = 1;
pathfoll.startat = 'path_ex_2';
[x,y,s] = bvpsuite2( 'bvps2_pathfoll_probdef', 'bvps2_pathfoll_settings',
    [], pathfoll );
```

With these lines, 5 plots, one at the value 0 and two each at 5 and 10, before and after the turning point, are displayed. As mentioned above, whether the plots are shown or not, can be turned off and on in the lines at the beginning of the file `pathfollowing.m`. The data to this plot can be called and plotted by the lines of code

```
load( [ 'path to the directory /' 'path_ex_3' ] )
figure
for ii=1:5
    sol=p_exact{1,ii}; subplot(1,5,ii); plot(sol.x1,sol.valx1)
    ylim([-0.4,0]); title(['\lambda_p=' num2str(p_exact{2,ii})])
end
```

The results of this computation are displayed Figure 12.

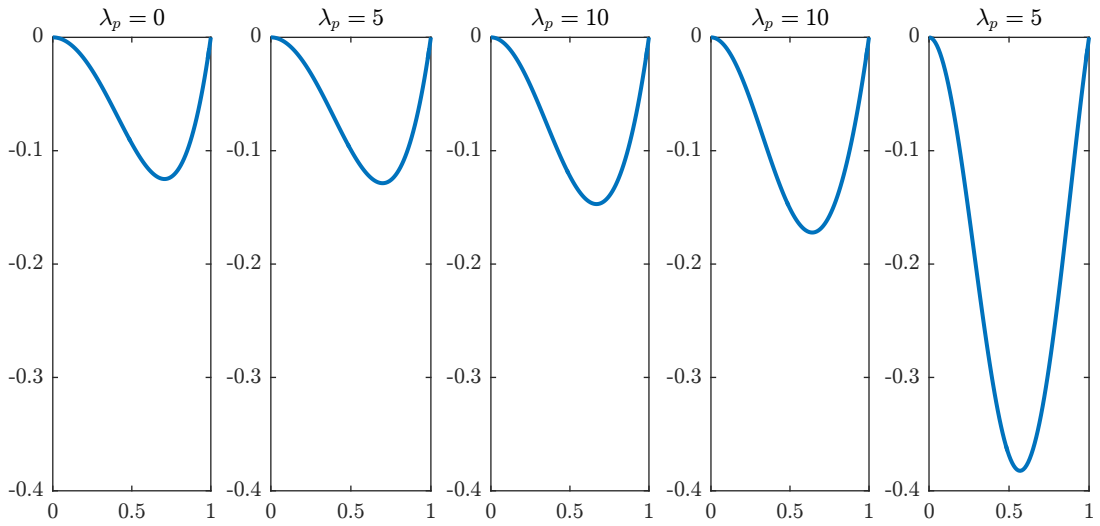

Figure 12: Pathfollowing problem (10): Solution of the system (10) for $\lambda_p = 0, 5, 10$ along the path displayed in Figure 11.

Instead of the evolution of the value of the parameter p under variation of λ^*, we want

to see the evolution of $z_1(\frac{1}{2})$ and $\|z_1\|_\infty$. Also, we want to still see the evolution of the parameter p. Therefore, we create a new file (`Home`→`New`→`Script`) containing the lines

```
function ret = PathCharData2(x1,coeff,ordnung,rho)
help = coeffToValues( coeff , x1 , ordnung , rho , 0:1/1000:1 , 0 );
ret= [ help(501); coeff(end-1); max(abs(help)) ];
end
```

and save it to the current folder under the name `PathCharData2.m`. Then, `bvpsuite2.0` is called by executing the lines

```
clear('pathfoll')
pathfoll.pathdata = @PathCharData2;
pathfoll.counter = 1;
pathfoll.startat = 'path_ex_3';
[x,y,s] = bvpsuite2( 'bvps2_pathfoll_probdef', 'bvps2_pathfoll_settings',
    [], pathfoll );
```

One more pathfollowing step is performed. A first prompt is displayed in the command window, asking whether to go back some steps or not, which can be answered by simply pressing the `enter`-key. The second prompt is displayed and can be answered by pressing `s` and then `enter`, in order to save the newly computed results. Along with the prompts, the plots are displayed and now three plots are following the evolution of a characteristic value of the solution. These plots are from top to bottom the evolution of

- $z_1(\frac{1}{2})$,
- p, and
- $\|z_1\|_\infty$, respectively.

Having saved the new path, these plots can be called by the codelet

```
load( [ 'path to the directory /' 'path_ex_4' ] )
lambdaValues = cell2mat(p_save(2,:)); % row-vector
charValues = cell2mat(p_save(3,:)); % 3-row matrix
yAxisLabel = {'$z_1(\frac{1}{2})$','$p$','$\|z_1\|_{\infty}$'};
figure
for ii=1:3
    subplot(1,3,ii); plot(lambdaValues,charValues(ii,:),'b-*')
    xlabel('$\lambda^*$'); ylabel(yAxisLabel(ii));
end
```

The predictor and corrector steps can be plotted analogously to the first solution plot in Section 3.7, where the main difference is that the value saved as `d` above is now a 3-row matrix instead of being a simple row-vector. The results are displayed in Figure 13. This goes to show, that whenever a path is saved, all settings can be adjusted and the pathfollowing resumed from the last computed position. In the same way, the solver settings concerning pathfollowing can be adjusted by setting any of the following lines

```
pathfoll.thetaMax            = % ajusted number
```

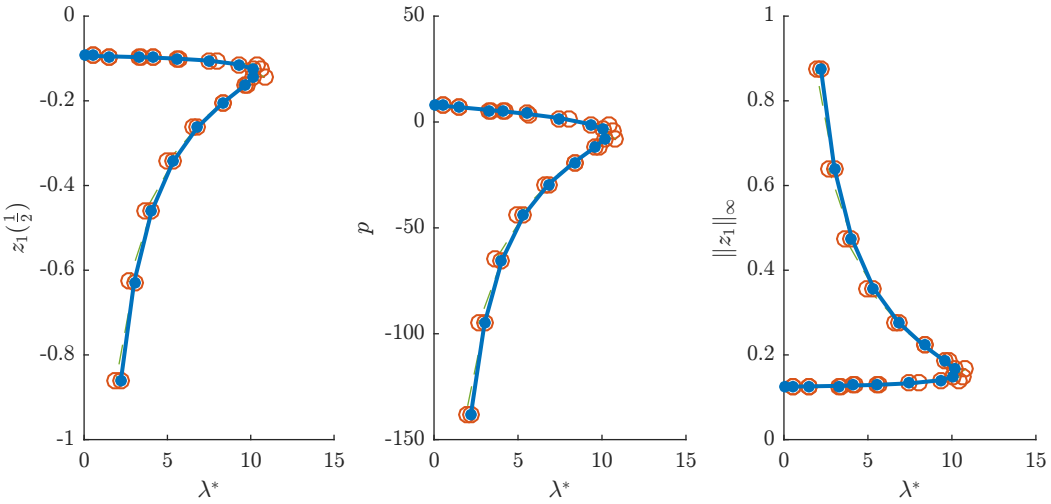

Figure 13: Pathfollowing problem (10): Evolution of the characteristic values $z_1(\frac{1}{2})$, p and $\|z_1\|_\infty$ respectively under variation of λ^*.

```
pathfoll.maxCorrSteps         = % ajusted number
pathfoll.maxSteplengthGrowth  = % ajusted number
pathfoll.angleMin             = % ajusted number
pathfoll.PredLengthFactor     = % ajusted number
pathfoll.CorrLengthGrowth     = % ajusted number
pathfoll.meshFactorMax        = % ajusted number
```

Hint: A setting that has not been discussed, is the field `max_pred_length`. When setting a value for this option, then the length of the predictor step in the plot of the characteristic value whose path is followed, will always be restricted to at most this value. During the run, when the prompt appears, by typing `p` and hitting the enter-key, a value can be set for the following steps for this option.

A Boundary conditions inside the interval

Remark: In `'c'` the array of points of the interval, in which boundary conditions are imposed, must be given. Also, in `'BV'` and `'dBV'`, the implicit form of the boundary conditions must be entered as detailed in Section 2.3.1.
Example: The following example was taken from [6].
Consider the linear singular BVP

$$C_1\Big(z_1(t) - z_2(t)\Big) - C_2 z_1''(t) + C_3 z_3'(t) = 0, \tag{11a}$$

$$C_1\left(z_1'(t) - z_2'(t)\right) + C_2 z_2^{(3)}(t) + C_4 z_3''(t) - C_5 = 0, \tag{11b}$$

$$C_6\left(C_1\left(z_1'(t) - z_2'(t)\right) + C_2 z_2^{(3)}(t)\right) + z_3(t) = 0, \quad \text{for } t \in [-D/2, D/2]\,, \tag{11c}$$

$$z_1(\pm D/2) = \pm B_1, \quad z_2(\pm D/2) = \pm B_2, \quad z_3(\pm D/2) = 0, \quad z_2''(0) = 0, \tag{11d}$$

where $C_1 = 4 \times 10^7$, $C_2 = 5 \times 10^{-12}$, $C_3 = -1.6 \times 10^{-3}$, $C_4 = 5.7 \times 10^{-2}$, $C_5 = -5 \times 10^2$, $C_6 = 10^{-15}$, $B_1 = 0.2$, $B_2 = 0.15$ and $D = 10^{-5}$.
In the file `bvps2_bcint_probdef.m`, we define our problem with the notation introduced in Section 2.3.1. We have

$$\boldsymbol{n} = 3, \quad \boldsymbol{l} = [2,3,2], \quad \boldsymbol{f} := \begin{pmatrix} C_1\left(z_1^{(0)}(t) - z_2^{(0)}(t)\right) - C_2 z_1^{(2)}(t) + C_3 z_3^{(1)}(t) \\ C_1\left(z_1^{(1)}(t) - z_2^{(1)}(t)\right) + C_2 z_2^{(3)}(t) + C_4 z_3^{(2)}(t) - C_5 \\ C_6\left(C_1\left(z_1^{(1)}(t) - z_2^{(1)}(t)\right) + C_2 z_2^{(3)}(t)\right) + z_3^{(0)}(t) \end{pmatrix},$$

$$\frac{\partial f_1}{\partial z_1^{(0)}(t)} = C_1, \quad \frac{\partial f_1}{\partial z_1^{(2)}(t)} = C_2, \quad \frac{\partial f_1}{\partial z_2^{(0)}(t)} = -C_1, \quad \frac{\partial f_1}{\partial z_3^{(1)}(t)} = C_3,$$

$$\frac{\partial f_1}{\partial z_1^{(1)}(t)} = C_1, \quad \frac{\partial f_1}{\partial z_2^{(1)}(t)} = -C_1, \quad \frac{\partial f_1}{\partial z_2^{(3)}(t)} = C_2, \quad \frac{\partial f_1}{\partial z_3^{(2)}(t)} = C_4,$$

$$\frac{\partial f_1}{\partial z_1^{(1)}(t)} = C_1 C_6, \quad \frac{\partial f_1}{\partial z_2^{(1)}(t)} = -C_1 C_6, \quad \frac{\partial f_1}{\partial z_2^{(3)}(t)} = C_2 C_6, \quad \frac{\partial f_1}{\partial z_3^{(0)}(t)} = 1,$$

$$\boldsymbol{g} := \begin{pmatrix} z_1^{(0)}(D/2) + B_1 \\ z_1^{(0)}(-D/2) - B_1 \\ z_2^{(0)}(D/2) + B_2 \\ z_2^{(0)}(-D/2) - B_2 \\ z_3^{(0)}(D/2) \\ z_3^{(0)}(-D/2) \\ z_2^{(2)}(0) \end{pmatrix},$$

$$\frac{\partial g_1}{\partial z_1^{(1)}(D/2)} = 1, \quad \frac{\partial g_2}{\partial z_1^{(1)}(-D/2)} = 1, \quad \frac{\partial g_3}{\partial z_2^{(0)}(D/2)} = 1, \quad \frac{\partial g_4}{\partial z_2^{(0)}(-D/2)} = 1, \quad \frac{\partial g_5}{\partial z_3^{(1)}(D/2)} = 1,$$

$$\frac{\partial g_6}{\partial z_3^{(1)}(-D/2)} = 1, \quad \frac{\partial g_7}{\partial z_2^{(1)}(0)} = 1.$$

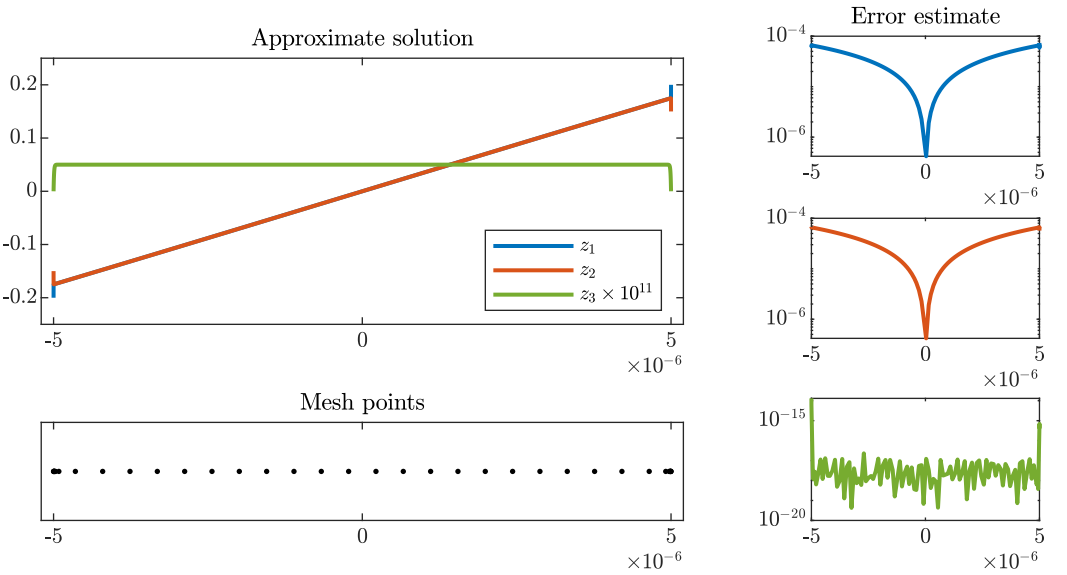

Figure 14: Problem (11) with boundary conditions inside the problem interval: Plot of the approximate solution, wherez_3 was multiplied with 10^{11}, the grid point distribution (every seventh point is displayed), and the error estimate.

The solver settings are specified in `bvps2_bcint_settings.m`. We chose to find the approximation to the solution of this BVP using a starting mesh with 101 equidistant mesh points and Gauss collocation with 2 collocation points in-between each two mesh points. The relative and absolute tolerances for the non-linear solver were set to 10^{-6} and to 10^{-4} for the absolute and relative tolerance of the mesh adaptation.
The following code lines can be executed to compute this example and plot the approximate solution and error estimate.

```
[x,y,s] = bvpsuite2( 'bvps2_bcint_probdef', 'bvps2_bcint_settings' );
plot( x, [ y(1:2, :); 1e11* y(3, :) ]); % approximate solution plot
figure; semilogy( s.x1tau, abs(s.errest) ) % plot of the error estimate
```

Note that the solution for z_3 is multiplied by 10^{11} in order to display the scaled computed solution of z_3 with z_1 and z_2. Also due to the different scaling of these functions, MATLAB may throw out some warnings about ill conditioned matrices, but should not affect the results of the computations too badly, depending on the accuracy the solutions are needed. If the computed approximations revealed themselves to be good enough in spite of the warnings, it may be an option to shut the warnings down, by the command

```
warning('off','MATLAB:nearlySingularMatrix');
```

This should be used with caution, since the warnings can very well point out some errors in the problem definition, solver settings or hint to another significant problem.
After some mesh adaptation, the final mesh contains 260 mesh points. The approximate solution, points distribution and error estimate are displayed in Figure 14.

References

[1] G. Kitzhofer. *Numerical treatment of implicit singular BVPs*, Ph.D. Thesis, Institute for Analysis and Scientific Computing, Vienna Univ. of Technology, Austria (2013)

[2] M. Schöbinger. *A new version of a collocation code for singular BVPs: Nonlinear solver and its application to m-Laplacians*, Master Thesis, Institute for Analysis and Scientific Computing, Vienna Univ. of Technology, Austria (2015)

[3] S. Wurm, `bvpsuite2.0` - *A new version of a collocation code for singular BVPs, EVPs and DAEs*, Master Thesis, Vienna Univ. of Technology, Vienna, Austria (2016)

[4] M. Fallahpour, Pathfollowing with automatic step-length control implemented in the new Matlab package bvpsuite2.0, Master Thesis, Vienna Univ. of Technology, Vienna, Austria (2020)

[5] R. März, O. Koch, D. Praetorius and E. B. Weinmüller, *Collocation methods for index-1 DAEs with a critical point*, Oberwolfach Report No. 18, ID 06016, of the Workshop on Differential-Algebraic Equations, Germany, pp. 81–84 (2006)

[6] M. Fallahpour, S. McKee and E. B. Weinmüller, *Numerical simulation of flow in smectic liquid crystals*, Appl. Numer. Math. 132, pp. 154–162 (2018)

[7] B. Higgins and B. Housam, *A simple method for tracking turning points in parameter space*, Journal of Chemical Engineering of Japan 43, pp. 1035–1042 (2010)

[8] G. Pulverer, G. Söderlind, E. B. Weinmüller, *Automatic grid control in adaptive BVP solvers*, Numer. Algorithms 56, pp. 61–92 (2011)

[9] F. D. Hoog and R. Weiss, *Collocation methods for singular boundary value problems*, SIAM J. Numer. Anal. 15, pp. 198–217 (1978)

[10] F. D. Hoog and R. Weiss, *The application of Runge-Kutta schemes to singular initial value problems*, Math. Comp. 44, pp. 93–103 (1985)

[11] E. B. Weinmüller, *Collocation for singular boundary value problems of second order*, SIAM J. Numer. Anal. 23, pp. 1062–1095 (1986)