TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DIPLOMARBEIT

A new version of a collocation code for singular BVPs: Nonlinear solver and its application to m-Laplacians

Ausgeführt am Institut für
Analysis and Scientific Computing
der Technischen Universität Wien

unter der Anleitung von
Ao. Univ. Prof. Dr. Ewa B. Weinmüller

durch
Markus Schöbinger, BSc.

Rennweg 61/2/24
1030 Wien

Wien, am August 14, 2015

Table of Contents

Abstract

This thesis documents the Matlab routine `bvpsuite`, developed at the Technical University of Vienna. It explains the ways `bvpsuite` solves nonlinear boundary value problems. These problems may be singular or posed on a semiinfinite interval. `bvpsuite` can also solve eigenvalue problems and problems containing unknown parameters. Before the `bvpsuite`-specific details an overview of Collocation and the Newton method is given.

This thesis also shows the changes applied to the routine during a resent update. The use of `bvpsuite` is illustrated by two fully documented examples, a linear and a nonlinear one. An overview of the settings options is also given.

The thesis ends with the application of `bvpsuite` to m-Laplacians, both before and after a smoothing transformation. The numerical rates of convergence are evaluated depending on the number of collocation points per collocation interval.

1 Introduction

1.1 Aim of this Thesis

The content of this thesis is centered around a collaboration with Stefan Wurm BSc. to update and rework `bvpsuite`, a code developed by the Institute for Analysis and Scientific Computing at the Technical University of Vienna.

This update was necessary because due to the updates which came with the newer Matlab versions the GUI and the symbolic toolbox for automatic formal derivations both received a massive overhaul, leading to instabilities and strong version dependencies in the code. Since both those elements were corner stones of the old `bvpsuite`, a complete restructuring of the code structure became necessary.

Since it was to be expected that future Matlab updates will again affect those two toolboxes, it was decided to abandon both concepts in the newer version. The new version of bypsuite is called strictly per console. The transformations for infinite intervals, which relied heavily on the symbolic toolbox, had to be rebuilt from scratch. The transformation is now computed on the fly during a function call via an interface between the calling function accessing the data file and the data file itself. An additional benefit of this new design is the possibility for the user to use his own transformation by feeding the relevant information into a file which can then be used by said interface.

In the course of this thesis we will first take a closer look at the solver for nonlinear systems, which was affected by the changes as well. We give an overview of its functionality followed by a detailed description of the single modules it is composed of, together with their calling sequence.

Furthermore we describe the new version of the code responsible for handling nonlinear problems, from the input to the solution, which the formerly mentioned solver is a part

of. This chapter contains a more practically orientated part dealing with the files which make up a specific problem and a description of the various parameters the user may manipulate. To make these concepts easier to grasp this chapter will end with two examples giving a step by step explanation how to preprocess problems to make them compatible with the code.

The last chapter will cover the results of an application of `bvpsuite` to a complex family of singular equations from the class of the so called m-Laplacians.

1.2 What is bvpsuite

`bvpsuite` is an open source code developed by members of the Institute for Analysis and Scientific Computing at the Technical University of Vienna.

The goal was to provide a code for solving boundary value problems (BVPs) of ordinary differential equations (ODEs) which can be applied to a large family of problems. `bvpsuite` is able to handle linear as well as nonlinear equations or systems of equations of arbitrary order, posed on finite or semi-infinite intervals. The problems may include singularities on one or both boundaries, computation of eigenvalues or other unknown parameters included[2].

The code is based on the technique of collocation. The solver for the arising nonlinear systems uses a combination of a Fast Frozen Newton and a Predictor-Corrector-search, backed up by built in Matlab functions in case both those two methods alone do not deliver the expected results.

The development of the code was partially funded by the Austrian Science Foundation, beginning in October 2004 to October 2006. Institute members involved in the development are Winfried Auzinger, Ernst Karner, Georg Kitzhofer, Othmar Koch, Gernot Pulverer, Christa Simon and Ewa Weinmüller (see link below).

The code as well as publications concerning this topic can be found at `http://www.asc.tuwien.ac.at/~ewa/software_development3.htm` .

2 The Solver

As already stated above, the solver is based on collocation utilizing a fast frozen Newton method as well as a Predictor-Corrector search. The aim of the following chapter is to give an overview of all these techniques and discuss how they are implemented in `bvpsuite`.

2.1 Collocation

The idea of collocation is to find a piecewise defined polynomial which satisfies the differential equation at a fixed set of points. More precisely:

Consider the problem

$$\begin{aligned} y'(t) &= f(t,y), \qquad t \in [a,b] \\ R(y(a), y(v)) &= 0 \end{aligned}$$

with a given function f and the boundary condition function R. Consider further a mesh on the interval $[a,b]$ given by $a := t_1 < t_2 < \cdots < t_N < t_{N+1} =: b$. On each subinterval $[t_i, t_{i+1}]$ define a finer mesh $t_i \leq \tau_1 < \tau_2 < \cdots < t_{p-1} < t_p \leq t_{i+1}$. The collocation solution is a polynomial y_i of degree p which satisfies $y_i'(t_j) = f(t, y_i(t_j))$ for all $1 \leq j \leq p$.

We additionally demand that the y_i be continuous when changing the subinterval, i.e. $y_i(t_{i+1}) = y_{i+1}(t_{i+1})$ for $1 \leq i \leq N-1$. Furthermore this piecewise polynomial is required to fulfill the boundary condition: $R(y_1(a), y_N(b)) = 0$.

Counting the equations there is one boundary condition, $N-1$ continuity conditions and p collocation conditions on each of the N subintervals, which nets a total of $N(p+1)$ conditions. Since the unknowns are the coefficients of N polynomials of degree $p+1$, these numbers match.

More difficult than adjusting the number of conditions to be posed is the question if the equations above actually can be solved. Especially in the general nonlinear case this is not easy to answer. The proof relies on quasilinearisiation and relation to Runge-Kutta methods. Details can be found in [5] and [1].

Up to now we have considered the problem to be given as a single ordinary differential equation of order one. It can however be easily extended to systems of equations by substituting y in the formulas above by a vector Y containing all the unknown functions and replacing the scalar $f(t,y)$ by a vector function $F(t,Y)$. Apart from these little notational differences the method itself stays exactly the same.

In order to use the method of collocation for higher order systems a common technique is to write each higher order equation as a system of first order equations via the following technique of substitutions:

$$y^{(k)} = f\left(t, y, y', \ldots, y^{(k-1)}\right) \quad \Leftrightarrow \quad \begin{aligned} z_1' &= z_2 \\ z_2' &= z_3 \\ &\vdots \\ z_{k-1}' &= z_k \\ z_k' &= f(t, z_1, z_2, \ldots, z_{k-1}, z_k) \end{aligned}$$

This enables the use of the collocation technique for any system of differential equations of any order. In the code we make use of an even more general formulation by allowing implicit equations, meaning $f(t, y(t), y'(t)) = 0$ for a given scalar function f. Again this can be extended to systems of higher order.

Using the collocation ansatz results in a nonlinear system of algebraic equations for the unknown coefficients. This system is then treated with one of the algorithms explained below.

2.2 Fast Frozen Newton

For the classical Newton method consider the problem of finding an x satisfying $f(x) = 0$ for some given function f. The aim is to find a sequence x_n which is easily computable and converges towards the exact solution x.

In the classical Newton method a starting point x_0 is expected to be given or chosen. The iteration then reads as

$$x_{n+1} := x_n - \frac{f(x_n)}{f'(x_n)}.$$

It can be shown that for f two times continuously differentiable with $f'(x) \neq 0$ at the exact solution there exists a domain of convergence around the solution x where the Newton iteration is defined and converges quadratically against the exact solution. For details see [6].

In higher dimensions the definition of the iteration is naturally extended as

$$x_{n+1} := x_n - DF^{-1}(x_n)F(x_n)$$

for the problem $F(x) = 0$. DF denotes the Frechet-derivative of F. Again it can be shown that for a starting approximation close enough to the exact solution the iteration converges.

This leaves two problems: The first is the necessity of a good starting approximation. While in `bvpsuite` the user can specify his own initial guess or use a solution of a related problem as a starting point, he cannot be expected to pinpoint the domain of convergence, which, depending on the problem, may be small.

This is the reason for another, more expensive method: If the Newton iteration finds itself not to deliver an improvement, it switches to alternative methods which have a larger domain of convergence. Once the convergence has stabilized, the Newton iteration takes over again.

The second problem is exclusive to the higher dimensional case. While in one dimension the division by the derivative is a simple computational operation, in higher dimensions the formula includes the inverse of a matrix, which is expensive to compute.

The second technique comes with the frozen aspect. The idea is to keep the matrix DF for a couple of steps without updating, which is only done when the convergence would come to a halt otherwise.

Since the Newton correction is computed on the basis of the LU-decomposition of DF, a fixed decomposition can be saved and be reused in multiple steps. Furthermore depending on the problem the construction of the matrix $DF(x_n)$ itself might be costly. This is especially true for problems on transformed intervals, where the data has to be preprocessed before each call of the problem file.

There exists another important technique to improve the classical Newton iteration, the so called damped Newton method. The modification is the introduction of a parameter $0 < \lambda \leq 1$ featured in a slightly modified update formula: $x_{n+1} = x_n - \lambda \Delta x_n$. The idea for this modification can be motivated by the following example:

Example: Consider the function $f = \arctan$. Of course the solution of $f(x) = 0$ is given by $x = 0$. However, if we start at $x_0 = 2$ and do one step of the classical Newton iteration, we arrive at $x_1 \approx -3.5$. The next step nets $x_2 \approx 14$ and it only gets worse from there. The problem is that the involved derivatives are too small and the iterations „shoot over“0 more and more. In this specific example it is clear that better results would be obtained if only a part of Δx_n would be added to the current step.

This observation can be generalized by the theorem found in [6] which states that if f is two times continuously differentiable and its Frechet-derivative being regular on a given domain surrounding the solution, then there exists a λ_{min} so that for every $\lambda \leq \lambda_{min}$ the sequence $||F(x_n)||$ is monotonically decreasing.

The damped Newton iteration utilizes this theorem by initializing λ as 1 and every time an update would lead to an increase in the norm of the function evaluation, λ is decreased by a predetermined factor and the step is tried again. If two consecutive steps could be done without any adjusting of λ, it is increased again.

In theory this algorithm will converge. It may however happen that λ becomes too small to be practical since the convergence speed is also dependent on λ. Therefore the algorithm terminates with an error message if the parameter falls below a given value. This will usually also occur if there is no nearby solution, such that the damped Newton iteration stalls near a local minimum of $||F(x)||$.

The Newton iteration used in `bvpsuite` is no classical damped method, however as we will see there are elements of the damped Newton in the algorithm, which is why it was reviewed here as well.

2.3 Predictor-Corrector Search

One problem with the damped Newton iteration discussed above is that evaluation for each λ requires the computation of F at a different point. Similarly as for DF, calls to F may be expensive depending on the structure of the problem. The aim of the Predictor-Corrector search is to „predict“the optimal value of λ from the previously gathered data, therefore saving expensive updates.

To be more precise, we consider the function

$$G(\lambda) := \left\| DF^{-1}(x_n) F(x_n - \lambda \Delta x_n) \right\|.$$

We aim to find a $\hat{\lambda}$ which minimizes the function G. Since G is itself highly nonlinear, we will approximate it by a quadratic polynomial by fixing the values $G(0)$, $G'(0)$ and $G(\lambda)$. In the last expression λ stands for the current value of λ.

At first it may seem strange to minimize G, when one really wants to minimize $||F(x_n)||$, or $||F(x_n + \lambda \Delta x_n)||$ respectively. The reason for this is that the Newton iteration itself is invariant under affine transformations: For any regular matrix A one will gain the exact same sequence x_n independent if the function F or AF is used in the problem statement. While both problems have the same solution, the norms of $F(x_n)$ and $AF(x_n)$ may differ drastically.

Here it is important to realize that the goal of the approximation is not to stop at a x_n with $||F(x_n)||$ being as small as possible, but instead one wants to terminate when $\Delta := ||x_n - x||$ is smaller than a given tolerance with x being the exact solution. Since the damped Newton iteration can be expected to converge linearly, the intermediate step sizes Δx_n can be used to measure the distance to the exact solution which is independent of the scaling of F.

The value $G(0)$ is known, because it corresponds to the norm of the previous correction Δx_{n-1}. Since we expect the current guess for the step Δx_n to be computed as well, this value is equal to $G(\lambda)$, which is therefore also known. This leaves the value $G'(0)$ to be determined.

To compute the derivative of G we will use the formula

$$\frac{d}{d\lambda} ||f(x)||^2 = 2 \langle f(x), f'(x) \rangle$$

and first compute the derivative of G^2 with respect to λ:

$$\frac{d}{d\lambda}\left\|DF^{-1}(x_n)F(x_n-\lambda\Delta x_n)\right\|^2 = 2\left\langle DF^{-1}(x_n)F(x_n-\lambda\Delta x_n), (DF^{-1}(x_n)F(x_n-\lambda\Delta x_n))'\right\rangle$$

Note that the factor $DF^{-1}(x_n)$ is independent of λ. Therefore we can apply the formula

$$\frac{d}{d\lambda}\left\langle a, b(\lambda)\right\rangle = \left\langle a, b'(\lambda)\right\rangle.$$

We will continue the computations only for the right hand side of the scalar product:

$$\begin{aligned}(DF^{-1}(x_n)F(x_n-\lambda\Delta x_n))' &= DF^{-1}(x_n)F'(x_n-\lambda\Delta x_n)\\ &= -DF^{-1}(x_n)DF(x_n-\lambda\Delta x_n)\Delta x_n\\ &= -DF^{-1}(x_n)DF(x_n-\lambda\Delta x_n)DF^{-1}(x_n)F(x_n)\end{aligned}$$

For the last manipulation note that Δx_n is given by $DF^{-1}(x_n)F(x_n)$. With this the derivative can be expressed as

$$\begin{aligned}&\frac{d}{d\lambda}\left\|DF^{-1}(x_n)F(x_n-\lambda\Delta x_n)\right\|^2 =\\ &-2\left\langle DF^{-1}(x_n)F(x_n-\lambda\Delta x_n), DF^{-1}(x_n)DF(x_n-\lambda\Delta x_n)DF^{-1}(x_n)F(x_n)\right\rangle.\end{aligned}$$

For $\lambda = 0$ we obtain

$$\begin{aligned}&\left\|DF^{-1}(x_n)F(x_n-\lambda\Delta x_n)\right\|^{2'}(0) =\\ &2\left\langle DF^{-1}(x_n)F(x_n), DF^{-1}(x_n)DF(x_n)DF^{-1}(x_n)F(x_n)\right\rangle =\\ &2\left\langle DF^{-1}(x_n)F(x_n), DF^{-1}(x_n)F(x_n)\right\rangle =\\ &2\left\|DF^{-1}(x_n)F(x_n)\right\|^2 = 2G(0)^2.\end{aligned}$$

Using the chain rule we can now compute the derivative of G itself:

$$\begin{aligned}G(\lambda) &= \left\|DF^{-1}(x_n)F(x_n-\lambda\Delta x_n)\right\| = \sqrt{\left\|DF^{-1}(x_n)F(x_n-\lambda\Delta x_n)\right\|^2}\\ G'(\lambda) &= \frac{\left\|DF^{-1}(x_n)F(x_n-\lambda\Delta x_n)\right\|^{2'}}{2\sqrt{\left\|DF^{-1}(x_n)F(x_n-\lambda\Delta x_n)\right\|^2}} = \frac{\frac{d}{d\lambda}G(\lambda)^2}{2G(\lambda)}\end{aligned}$$

This gives

$$G'(0) = \frac{G(0)^{2'}}{2G(0)} = \frac{-2G(0)^2}{2G(0)} = -G(0).$$

With this we can start identifying the interpolating polynomial. We make the ansatz $\tilde{G}(x) = ax^2 + bx + c$ with unknown coefficients a, b and c.

Clearly $c = G(0)$ and $b = G'(0)$ where $G(0)$ and $G'(0)$ are known. Only a needs to be determined. Since $G(\lambda)$ is known, we find

$$\begin{aligned} G(\lambda) &= a\lambda^2 + \lambda G'(0) + G(0) \\ a &= \frac{G(\lambda) - \lambda G'(0) - G(0)}{\lambda^2}. \end{aligned}$$

Minimizing the value of this polynomial is an elementary task. Of course this method can only succeed if the polynomial is indeed convex, meaning $a > 0$. We will not give a proof of this property here.

$$\begin{aligned} \tilde{G}(x) &= ax^2 + bx + c \\ \tilde{G}'(x) &= 2ax + b \\ 0 &= 2ax + b \\ x &= -\frac{b}{2a} = -\frac{\lambda^2 G'(0)}{2(G(\lambda) - \lambda G'(0) - G(0))} \end{aligned}$$

The x calculated this way is then used as a new guess for λ. As we will see in the code, in practice this new λ will be forced to be located in a special interval to prevent too big jumps or values which are not meaningful in the problem setting.

What we have discussed until now was is predictor step: We use the data from one previous try to predict a good value for a new λ. If in subsequent steps the new λ is also not found to deliver the expected results, the corrector step is used.

The corrector works similarly to the predictor in that it calculates an interpolating polynomial which is then minimized. In contrast to the predictor polynomial, however, the corrector polynomial is chosen to be of degree 3. It uses $G(0)$ and $G'(0)$ as well as $G(\lambda_1)$ and $G(\lambda_2)$ with λ_1 and λ_2 being the two previously tried values for λ.

We make the ansatz $\tilde{G}(x) = ax^3 + bx^2 + cx + d$. As in the predictor step, the last two coefficients are given by $d = G(0)$ and $c = G'(0)$. In the code the other two coefficients are computed according to

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\lambda_1 - \lambda_2} \begin{pmatrix} \frac{1}{\lambda_1^2} & -\frac{1}{\lambda_2^2} \\ -\frac{\lambda_2}{\lambda_1^2} & \frac{\lambda_1}{\lambda_2^2} \end{pmatrix} \begin{pmatrix} G(\lambda_1) - \lambda_1 G'(0) - G(0) \\ G(\lambda_2) - \lambda_2 G'(0) - G(0) \end{pmatrix}.$$

To understand this formula we will first calculate the tailor expansion of G up to the cubic terms: $G(x) = G(0) + G'(0)x + \frac{G''(0)}{2}x^2 + \frac{G'''(0)}{6}x^3 + \mathcal{O}(x^4)$. If we substitute G by the interpolating polynomial $\tilde{G}$, the unknown last term will disappear, since $\tilde{G}$ is cubic. Using this the formula simplifies to

$$
\begin{aligned}
\begin{pmatrix} a \\ b \end{pmatrix} &= \frac{1}{\lambda_1 - \lambda_2} \begin{pmatrix} \frac{1}{\lambda_1^2} & -\frac{1}{\lambda_2^2} \\ -\frac{\lambda_2}{\lambda_1^2} & \frac{\lambda_1}{\lambda_2^2} \end{pmatrix} \begin{pmatrix} \frac{\tilde{G}''(0)}{2}\lambda_1^2 + \frac{\tilde{G}'''(0)}{6}\lambda_1^3 \\ \frac{\tilde{G}''(0)}{2}\lambda_2^2 + \frac{\tilde{G}'''(0)}{6}\lambda_2^3 \end{pmatrix} \\
\begin{pmatrix} a \\ b \end{pmatrix} &= \frac{1}{\lambda_1 - \lambda_2} \begin{pmatrix} \frac{\tilde{G}''(0)}{2} + \frac{\tilde{G}'''(0)}{6}\lambda_1 - \frac{\tilde{G}''(0)}{2} - \frac{\tilde{G}'''(0)}{6}\lambda_2 \\ -\frac{\tilde{G}''(0)}{2}\lambda_2 - \frac{\tilde{G}'''(0)}{6}\lambda_1\lambda_2 + \frac{\tilde{G}''(0)}{2}\lambda_1 + \frac{\tilde{G}'''(0)}{6}\lambda_1\lambda_2 \end{pmatrix} \\
\begin{pmatrix} a \\ b \end{pmatrix} &= \frac{1}{\lambda_1 - \lambda_2} \begin{pmatrix} \frac{\tilde{G}'''(0)}{6}\lambda_1 - \frac{\tilde{G}'''(0)}{6}\lambda_2 \\ -\frac{\tilde{G}''(0)}{2}\lambda_2 + \frac{\tilde{G}''(0)}{2}\lambda_1 \end{pmatrix} \\
\begin{pmatrix} a \\ b \end{pmatrix} &= \frac{1}{\lambda_1 - \lambda_2} \begin{pmatrix} (\lambda_1 - \lambda_2)\left(\frac{\tilde{G}'''(0)}{6}\right) \\ (\lambda_1 - \lambda_2)\left(\frac{\tilde{G}''(0)}{2}\right) \end{pmatrix} \\
\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} \frac{\tilde{G}'''(0)}{6} \\ \frac{\tilde{G}''(0)}{2} \end{pmatrix}.
\end{aligned}
$$

Therefore the so defined values a and b are indeed the missing coefficients of the interpolating polynomial. Again we will differentiate this polynomial in order to find the minimum:

$$
\begin{aligned}
\tilde{G}(x) &= ax^3 + bx^2 + cx + d \\
\tilde{G}'(x) &= 3ax^3 + 2bx + c \\
0 &= 3ax^3 + 2x + c \\
0 &= \frac{3}{2}ax^3 + bx + \frac{c}{2} \\
x &= \frac{-b + \sqrt{b^2 - 3ac}}{3a}
\end{aligned}
$$

In the last step the solution formula for quadratic equations was used. This value will then be used as the next λ. The decision to use the plus sign in the solution formula instead of the minus sign was reached by the author of the original code without giving any motivation or references.

2.4 Overview of solve-nonlinear-sys

In this section we will give an overview over the solver file itself. Since it is rather lengthy and contains a lot of technical details, we will only focus on the parts which are crucial in understanding the functionality of the solver.

The first couple of lines reads in parameters from the settings file like the chosen value of λ_{min} or the maximum number of function evaluations. One of these deserves special mention:

```
lsq = 0;
```

`lsq` is a switch which can take the values 0 or 1 and decides which Matlab internal routine is called if both algorithms given above do not achieve convergence. Note that this switch is not included in the settings file as it is intended only for advanced users and as a last resort if the default setting does not work.

Per default `lsq` is set to 0, which means that the routine `fsolve` is called. The aim of fsolve is also to solve the equation $f(x) = 0$ for a given function f. It supports a variation of parameters including initial guesses and the derivative of f. A complete documentation can be found at `http://de.mathworks.com/help/optim/ug/fsolve.html`.

The alternative corresponding to `lsq` equal to 1 is `lsqnonlin`. In contrast to `fsolve`, the aim is not explicitly to solve for $f(x) = 0$, but rather to minimize $||f(x)||$. This minimum does in general not necessarily have to be equal to 0, however `lsqnonlin` has proven to perform in many cases comparable to `fsolve` and both algorithms have been known to converge against points which are not strictly speaking solutions of the problem. Since the results are comparable, `lsqnonlin` is an alternative if `fsolve` does not perform adequately. A complete documentation is given at `http://de.mathworks.com/help/optim/ug/lsqnonlin.html`.

```
[TolFactor, DOC, new_x, U, L, G0, G, F, delta_x, simplified_delta_x, ...
    fcount, logstruct] = ...
     determine_position2(Fhandle, DFhandle, x, bvpfile, tau, settings, ...
        psival, psi, lambdamin, fcount, logstruct, [], [], rho);


if TolFactor < 1
   if display
      fprintf('\n\n Tolerances satisfied\n');
   end
   return
end
```

This is the first function call concerning the actual solver. The subroutine `determine_position2` calculates a first Newton step and uses the residual to decide how the solver should proceed. Since the code of `determine_position2` has many similarities to the one of the fast frozen Newton version, we will not take a closer look at it here.

Out of the list of output parameters we should note `TolFactor`, which contains the factor by which the residual is still too large for the solution to be able to fulfill the tolerances. As we can see, if this factor happens to be smaller than one, the solution is assumed to be precise enough and the solver terminates.

`DOC` is the essential decision making variable. It takes the value 1, 2 or 3 and decides, which algorithm is used next. 1 corresponds to the fast frozen Newton technique, 2 to the predictor-corrector algorithm and 3 means that the backup Matlab routines are used.

Two other variables involved in the decision making below are `G` and `G0`, which contain the norm of the current residual and the previous residual respectively.

```
while 1

...

if itcount > max_iter
   error(' Maximum number of iterations exceeded');
end

if fcount > max_F_evals
   error(' Maximum number of function evaluations exceeded');
end

switch DOC
```

Here we see the main loop of the solver. Depending on the variable DOC, one of the three algorithms is called until either one of them decides the tolerances to be satisfied or an error is encountered.

First we will take a closer look at case 1, which is the fast frozen Newton.

```
      UpdateJac = G > updateJacFactor * G0;

    if UpdateJac

        DF = feval(DFhandle,bvpfile,x,tau,psival,psi,rho);

        [L,U]=lu(DF);

        delta_x = U\(L\(- F));

        G0 = norm(delta_x);

    else
        delta_x = simplified_delta_x;
        G0 = G;
    end
```

Leaving out the logging calls, this is the beginning of case 1. First it is checked if the norm of the residual has improved enough to justify continuing without updating the Jacobian. In this case, the old values are used as the basis for the following calculations. Otherwise the Jacobian is evaluated at the current position. Then the LU-decomposition is calculated using the Matlab routine `lu`. Then the residual and its norm are computed.

```
    new_x(:) = x(:) + delta_x;

    new_F = feval(Fhandle,bvpfile,new_x,tau,psival,psi,rho);

    simplified_delta_x = U\(L\(-new_F));

    G = norm(simplified_delta_x);
```

Again omitting the non essential details here we see the application of the Newton correction. The other three lines evaluate F at the new position and perform a virtual second

Newton step to check the rate of improvement.

```
    if G < (1-lambdamin/2) * G0

       TolFactor = ...
          check_tolerances(new_x,simplified_delta_x,AbsTol,RelTol);

       if TolFactor < 1

           new_x(:) = new_x(:) + simplified_delta_x;

           return
       end

       DOC = 1;

       x = new_x;
       F = new_F;

    elseif UpdateJac == 0
       DOC = 1;

    else
       DOC = 2;

    end
```

Finally the algorithm checks if there was a significant improvement in the norm of the residual. If so, it also checks if the tolerances are satisfied and terminates. Otherwise the new approximation is accepted and the solver continues to use the fast frozen Newton method.

If there was no significant improvement, there are two cases to consider: If the Jacobian has not been updated, the fast frozen Newton is given another try in which it is updated. If it already was, the solver switches to the second branch of the main loop containing the predictor-corrector method, which we will now look into in more detail.

```
Accept_Lambda = G < (1-lambdamin/2)*G0;
nCorrections = 0;

while ~Accept_Lambda

   if nCorrections == 0
      l2 = lambda;

      Gprime0 = -G0;
      lambda = - lambda^2 * Gprime0 / (2* (G - G0 - lambda * Gprime0));

      lambda = max(l2/10 , min(lambda, 1));

      l1 = lambda;

      nCorrections = 1;
```

```
else

   coeff = 1/(l1-l2) * [1/l1^2 -1/l2^2 ; -l2/l1^2 l1/l2^2] * ...
                       [G1 - Gprime0*l1 - G0 ; G2 - Gprime0*l2 - G0];

   ca = coeff(1);
   cb = coeff(2);

   lambda = (-cb + sqrt(cb^2-3*ca*Gprime0))/(3*ca);

   lambda = max(l1/10 , min(lambda, l1/2));

   l2 = l1;
   l1 = lambda;

   nCorrections = nCorrections + 1;
end

if lambda < lambdamin
   DOC = 3;
   break;
end
```

This branch also starts by checking if there already is a satisfactory rate of convergence. While this cannot be the case if the solver has just switched there from the first branch, it could happen if it later decides to stay in the second branch.

If the current value of λ does not get accepted the algorithm first enters the predictor branch which uses the quadratic interpolation technique discussed above. Afterwards it is made sure that the new value of λ is at least a tenth of the old λ and not larger than 1.

If the predictor alone is not enough, the corrector step is initiated where we again see the already discussed formulas as well as the assurance that the new value of λ stays in a reasonable vicinity of the old one.

Finally, if after several corrector steps λ is still not satisfactory but already smaller than the given minimum value, the solver switches to the Matlab routines.

After each of those conditional branches the Newton step corresponding to the calculated λ is performed and checked for improvement. Since this is largely similar to the calculations in case 1, we will omit the technical details of this operations.

```
   if Accept_Lambda
x = new_x;

if G < switchToFFNFactor * G0
   DOC = 1;
else
   DOC = 2;
end
```

```
end

if DOC ~= 3
   TolFactor = check_tolerances(x,simplified_delta_x,AbsTol,RelTol);


   if TolFactor < 1
      new_x(:) = x(:) + simplified_delta_x;

      return
   end
end
```

If a λ gets accepted, the algorithm decides if the improvement was good enough to justify switching back to the fast frozen Newton routine or if it should stay in the predictor-corrector branch. In the case that it did not decide to fall back to the Matlab routines it is also checked if the current solution is accepted.

The rest of the branch are initializations for the case that the solver stays there. As there is nothing to be learned about the nature of the solver there, we will not go into further details.

Lastly we will have a look at the third branch. The essential calls are contained in the listing below.

```
if(~TRM)
  disp('tried to use TRM. Solution may be imprecise or wrong');
  return;
end

options = ...
    optimset('Display','off','MaxIter',max_iter,'MaxFunEvals',max_F_evals);

options.TolX = sqrt(max(RelTol,AbsTol));
options.TolFun = 0;
options.Jacobian = 'on';
options.LargeScale = 'on';
switch nPreviousTRMIterates
   case 0
      options.TolX = sqrt(max(RelTol,AbsTol));
   case 1
      options.TolX = max(RelTol,AbsTol);
   case 2
      options.TolX = 1000*eps;
   case 3
      options.Algorithm =  'levenberg-marquardt';
   case 4
      fprintf('Zerofinder cannot fulfill the given tolerances');
      return;
end

options.Display = 'iter';
options.MaxIter = max_iter - itcount;
```

```
options.MaxFunEvals = max_F_evals - fcount;

if lsq
    [x,~,F,ExitFlag,lsqLog,~,DF] = ...
     lsqnonlin(FDFhandle,x,[],[],options,bvpfile,tau,psival,psi,rho);
else
    [x,F,ExitFlag,lsqLog,DF] = ...
      fsolve(FDFhandle,x,options,bvpfile,tau,psival,psi,rho);
end
```

If the user has decided against using the Matlab routines, the algorithm returns the current solution together with the warning that it will likely not fulfill the given tolerances. Otherwise a set of options for the Matlab routines is created. Out of the settings, the only one which requires further explaining is the given tolerance. Note that the variable `nPreviousTRMIterates` contains the number of times the solver has already chosen the third branch.

In the first try the tolerances are set to the square root of the problem tolerances, as in the best case the Matlab routines only need to provide a new initial guess from where the other algorithms can overtake again. Should this alone not work, it is then tried with the actual tolerances and then with one thousand times double precision. Note that `eps` is a predefined Matlab variable which is equal to the smallest possible relative value expressible in double precision.

As a last resort the Levenberg-Marquardt algorithm is used. If all tries above should fail, the solver terminates with an error message.

The rest of the branch consists of error handling if the Matlab routines did not converge or converged against a point which is not a solution. In the case of a successful call the solver again uses the subroutine `determine_position2` to decide on a further course of action.

3 Changelog

This chapter deals with the changes to `bvpsuite`, both on the code level and from the view of a user. First we will cover the code responsible for the processing of nonlinear equations. We will then go on to discuss the various parameters one can set in the input files to customize the calculations. In the end of this chapter we will have a look at two specific examples for the purpose of illustration.

3.1 Handling of Nonlinear Problems

Since the cornerstone of the routines for the processing of nonlinear problems - the file `solve_nonlinear_sys` - has already been discussed in Chapter 2, this chapter will focus on the functions the solver is embedded in.

The function called by the user is of course `bvpsuite2`, which takes the problem file, as well as the settings file and optionally a initial solution as parameters. `bvpsuite2` itself however is only responsible for a basic preprocessing of the given data as well as managing the calls to the modules. This section will focus on the module `solveNonLinearProblem`, which is responsible for the handling of nonlinear equations.

The header of `solveNonLinearProblem` reads as follows:

```
function [x1tau,valx1tau,solStruct] = solveNonLinearProblem(problem, ...
    settings, x1, rho, initProfile, transformInitProfile)
```

Starting with the left hand side the function returns first the grid the solution is defined on and second the calculated function values on those grid points. The `solStruct` structure contains additional information about the solution, as can be seen later on. All of these parameters are handed to `bvpsuite2` and returned to the user.

We now discuss the input parameters in order:

- `problem` is the problem file (also referred to as bvp file).
- `settings` is the settings file.
- `x1` contains the grid on which the solution is to be calculated.
- `rho` is an integer giving the number of collocation points per interval.
- `initProfile` is structure allowing the user to choose a specific starting solution. The syntax to create such a structure will be covered in the section covering the problem file. It should be noted that the solution structure can be used as an initial profile. This means that it is possible to use the solution of one equation as the initial profile of another one without having to change any files. Should no initial profile be given by the user, a function with a constant value 1 will be used.
- `transformInitProfile` acts as a boolean switch, which is only needed for problems posed on semiinfinite intervals. It tells, if the initial grid is already transformed to

the interval the solution will be calculated on. The value 1 means that the initial profile is defined on the interval given by the bvp file (which could be semiinfinite) while 0 means it is already defined on the transformed grid used for computations (in the semiinfinite case this will usually be the interval $[0, 1]$). Note that for a problem on a finite interval with no transformation given this flag will not change the further course of computation.

The last parameter
After the call the function initializes a few variables for later use:

```
m=length(rho);
n = feval_problem(problem,'n');
N=length(x1)-1;
h(1:N)=x1(2:N+1)-x1(1:N);
tau(1:N,1:m)=x1(1:N).'*ones(1,m)+h(1:N).'*rho(1:m);
ordnung = feval_problem(problem,'orders');
parameter = feval_problem(problem,'parameters');
```

```
p=zeros(parameter);
y=zeros(n,max(ordnung),N);
z=zeros(n,m,N);
```

Most of these lines should be self-explaining. The vector `h` stores the step lengths while `tau` is a matrix which stores per line all collocation points between one grid point and the next.

The last three calls are intended to improve performance. In Matlab it is more efficient to initialize a variable with zeros before filling it with data via loops, since without this initialization step the variable would change its dimensions with every loop iteration, causing internal recalculations and in the worst case relocations of memory.

```
if(nargin<6)
    transformInitProfile = 1;
end
```

This branch guarantees that the parameter `transformInitProfile` has a meaningful value even if it is not given explicitly. As stated above 1 is the default state in the sense that the initial Profile is stated on the same interval as the problem.

```
initProfile=initial_coefficients(problem, x1, initProfile, rho, ...
   transformInitProfile);
x0 = initProfile.initialCoeff;
solStruct.initProfile = initProfile;
```

The function `initial_coefficients` is responsible for transforming the initial profile, which is given as values at grid points, into coefficients of the collocation basis polynomials.

As can be seen the input parameters for `initial_coefficients` are quite similar to the ones of `solveNonLinearProblem`. It should be noted that using the same initial problem variable as input and output causes no loss of information, as the only field of the structure which gets altered is `initialCoeff`. The others are kept the same. This field, which is the only one used in the further computations, is saved to the local variable `x0`, while the whole structure is saved in the solution structure in case it is needed for future function calls.

A technical detail concerning the handling of the different grids: While the initial profile is posed on its corresponding initial grid, the solution is computed on the grid given by `x1`. Apart from the fact that both grids refer to the same interval, there is a priori no connection between these two grids, as the user has full freedom to choose both of them. During the calculation of the initial coefficients it may therefore be necessary to evaluate the initial profile at points which are not part of the initial grid. In those cases the Matlab function `interp1` is used, which gives a spline interpolation of the available data.

```
psi=zeros(m,max(ordnung)+m,max(ordnung));
for ord=1:max(ordnung)
    for i=1:m
        psi(i,1+max(ordnung)-ord:m+max(ordnung),ord)=Psi(i,rho,ord);
    end
end
psival=zeros(max(ordnung),m,m+2);
for ord=1:max(ordnung)
    for i=1:m
        %evaluation of psi
        psival(ord,i,1:m)=polyval(psi(i,:,ord),rho(1:m));
        psival(ord,i,m+1)=polyval(psi(i,:,ord),1);
        psival(ord,i,m+2)=polyval(psi(i,:,ord),0);
    end
end
```

This part of the code first gets the Matlab representations of the collocation basis polynomials and stores them in the variable `psi`. These vectors of coefficients are then used to evaluate those polynomials on the collocation points. Note that it is not necessary to evaluate them on all points stored in `tau`, since the values stay the same over each collocation interval.

```
coeff = ...
   solve_nonlinear_sys(@F,@DF,@FDF,x0,problem,x1,settings,psival,psi,rho);
```

Next comes the call of the cornerstone, the function `solve_nonlinear_sys`. Since it is extensively covered in chapter 2, we will not discuss any further.

```
%entries of "coeff" will be transformed to the names of the documentation
%y,z,p see manual!
j=0;
for i=0:N-1
    for q=1:max(ordnung)
        for ni=1:n
```

```
            if q<=ordnung(ni)
                j=j+1;y(ni,q,(i)+1)=coeff(j);
            end
        end
    end
    z(1:n,1:m,(i)+1)=reshape(coeff(j+1:j+n*m),n,m);
    j=j+n*m;
end
p(1:parameter)=coeff(j+1:j+parameter).';
```

As already stated in the comment to the code, the purpose of those loops is solely to make the data from the coefficient vector conformal to the nomenclature and the following formulas. For the referenced manual see [2].

```
valx1tau=zeros(n,N+N*m+1);
for i=1:n
   stelle=0;
   for j=0:N-1
       if ordnung(i)~=0
           if j~=0
               help=Poptimiert(0, j, i , x1((j)+1), m, x1, h, y, z, ...
                   psival, ordnung(i), m+2);
           else
               help=P(0,j,i,x1((j)+1),m,x1,h,y,z,psi,ordnung(i));
           end
       else
           help=P(0,j,i,x1((j)+1),m,x1,h,[],z,psi0,0);
       end
       stelle=stelle+1;valx1tau(i,stelle)=help;
       for k=1:m
           if ordnung(i)~=0
               help=Poptimiert(0, j, i, tau((j)+1,k), m, x1, h, y, z, ...
                   psival, ordnung(i), k);
           else
               help=P(0,j,i,tau((j)+1,k),m,x1,h,[],z,psi0,0);
           end
           stelle=stelle+1;valx1tau(i,stelle)=help;
       end
   end
   if ordnung(i)~=0
       help=P(0,N-1,i,x1((N)+1),m,x1,h,y,z,psi,ordnung(i));
   else
       help=P(0,N-1,i,x1((N)+1),m,x1,h,[],z,psi0,0);
   end
   stelle=stelle+1;
   valx1tau(i,stelle)=help;
end

stelle=0;
x1tau=zeros(1,N);
for j=0:N-1
     stelle=stelle+1;x1tau(stelle)=x1((j)+1);
     for k=1:m
```

```
            stelle=stelle+1;x1tau(stelle)=tau((j)+1,k);
        end
    end
    stelle=stelle+1;x1tau(stelle)=x1((N)+1);
```

This rather lengthy and technical part transforms the coefficient data back into the solution grid and its function values. Since aside from that functionality this piece of code does not handle any other manipulation and since it does not provide any deeper inside to copy the evaluation functions here, we will leave it at that.

From a practical point of view the last part is more interesting, as it shows the kinds of information the solution structure stores:

```
solStruct.x1 = x1tau(1:m+1:end);
solStruct.valx1 = valx1tau(:,1:m+1:end);
solStruct.x1tau = x1tau;
solStruct.valx1tau = valx1tau;
solStruct.parameters = p;
solStruct.coeff = coeff(1:end-length(p));
solStruct.initProfile = initProfile;
```

The first two lines feed the solution the grid and the calculated values on this grid. This may seem like a slight redundancy, as both values are already returned via the first two return arguments. The reason for this is to make the program more comfortable for users who are only interested in the solution itself and in no further data, as in that case the third return parameter does not have to be saved at all.

`x1tau` and `valx1tau` contain the collocation grid and the calculated function values on it. Since those values have to be calculated anyhow and the collocation grid is finer than x1 this can be seen as a free extension of the solution if it is not critical to have the values on precisely the `x1` grid.

The line concerning `solStruct.parameters` only triggers if the problem contains unknown parameters or the search for eigenvalues. In case of eigenvalue problems the eigenvalue is stored as the last parameter.

In case the user should be interested in the representation of the solution in the collocation basis (for example to calculate his own interpolations for further points), the coefficient vector is stored in the solution structure as well. Since the last entries of this vector correspond to the parameters which are already covered, they are left out.

3.2 List of Parameters

In this section we have a look at the various parameters a user can change via the settings file. Our starting point will be the file `default_settings`, which can be seen as a template to create own setting files. In case the user sees no need to manipulate any settings, this file is filled with a reasonable choice of parameters which are tested to work for a broad spectrum of equations.

```
function [ret] = default_settings(request)

switch request
```

Just like in the bvp file, which we will look at in the next section, the main body of the settings file is a `switch` statement. As we have seen in the second chapter, the retrieval of information from the files is handled via the Matlab command `feval`. In order for this to work the data files need to have the form of functions, which return the relevant data according to the received request.

Since every case-branch of the main switch statement corresponds to one piece of data stored in the file, we will discuss them one by one.

- ```
  case 'mesh'
      ret=linspace(0,1,100);
  ```

  The first customizable parameter is also the most important one for most applications. The mesh request returns the vector of grid points on which the solution is to be calculated. It must be emphasized that the interval of the mesh does not need to be the same as the interval the problem is posed on. This is most easily explained by looking at the lines in `bvpsuite2` responsible for the processing of the solution grid.

  ```
  x1 = feval(settings,'mesh');

  % ...

  interval = feval_problem(problem,'interval');
  x1 = (interval(2)-interval(1))/(x1(end)-x1(1))*(x1+x1(1))+interval(1);
  ```

  As can be readily seen in the preprocessing step the mesh is read from the settings file and then linearly mapped onto the problem interval. The idea behind this is that in application only the distribution of the grid points is relevant and if the user has his own routines to generate a distribution of points he is interested in, there is no need for him to transform them manually.

  Another, possibly even more important reason is that this choice of design enables a setting file independent of the parameters of a specific problem. This separation between problem data and settings data was a major concern during the development process. As the simplest consequence it is always an option to just use the default_settings file, regardless of the equation involved.

- ```
  case 'collMethod'
      ret='gauss';
  ```

  The second parameter concerns the location of the collocation points in each collocation interval. There are the three default options 'gauss', 'lobatto' and 'uniform'.

A visualization of these three options is given by Figure 1.

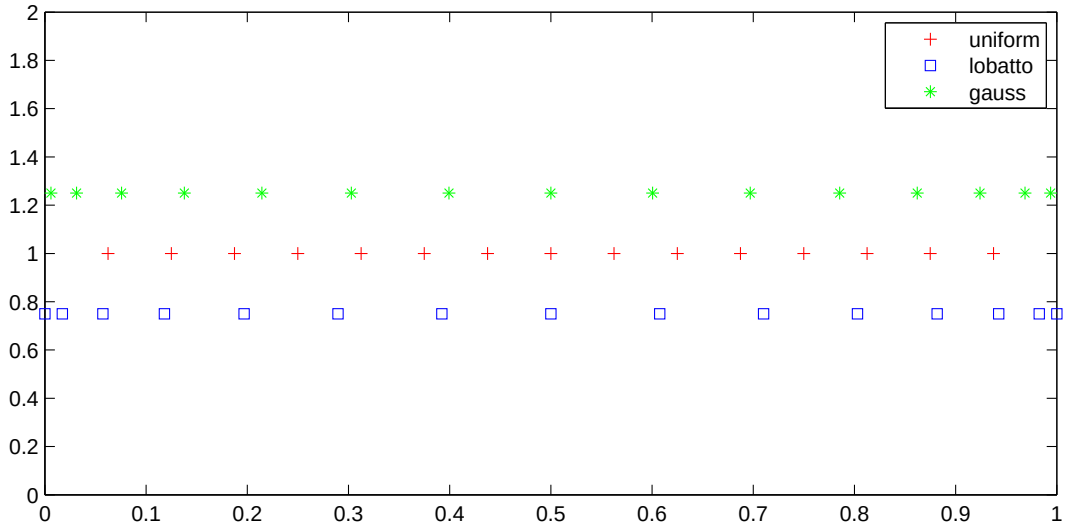

Figure 1: Different distributions of collocation points

There is also the option to write 'user' into the return statement. This makes it possible to choose a different distribution of one's own.

- ```
  case 'collPoints'
      ret=1;
  ```

  If 'user' was selected in the branch above, this is the place to put the desired distribution, posed as a vector. In this case the program expects these values to be in the closed interval $[0, 1]$. Note that for problems with singularities on one or both boundaries it might be a bad idea to place collocation points at zero or one. Out of the three predefinied options only the lobatto points make use of the interval boundaries.

  If one of the three coded options is selected in the collMethod branch, collPoints gives the number of points per interval. It should be noted that a uniform distribution works for any number of points, the Gauss and Lobatto options both support a maximum of 15 points per interval, with one point being excluded in the lobatto case. While 15 points per interval should be far more than enough for most problems, it is still possible to use for example twenty Gaussian points by switching to 'user' and writing the points into this return statement.

- ```
  case 'meshAdaptation'
      ret=0;
  ```

This parameter switches the mesh adaptation routine on or off. 1 toggles the mesh adaptation on, while 0 means no mesh adaptation. While using the mesh adaptation routine may slow down the computation in examples where it is not needed, it can be a great help if the smoothness of the solution is significantly varying.

Figure 2 illustrates the result of the mesh adaptation.

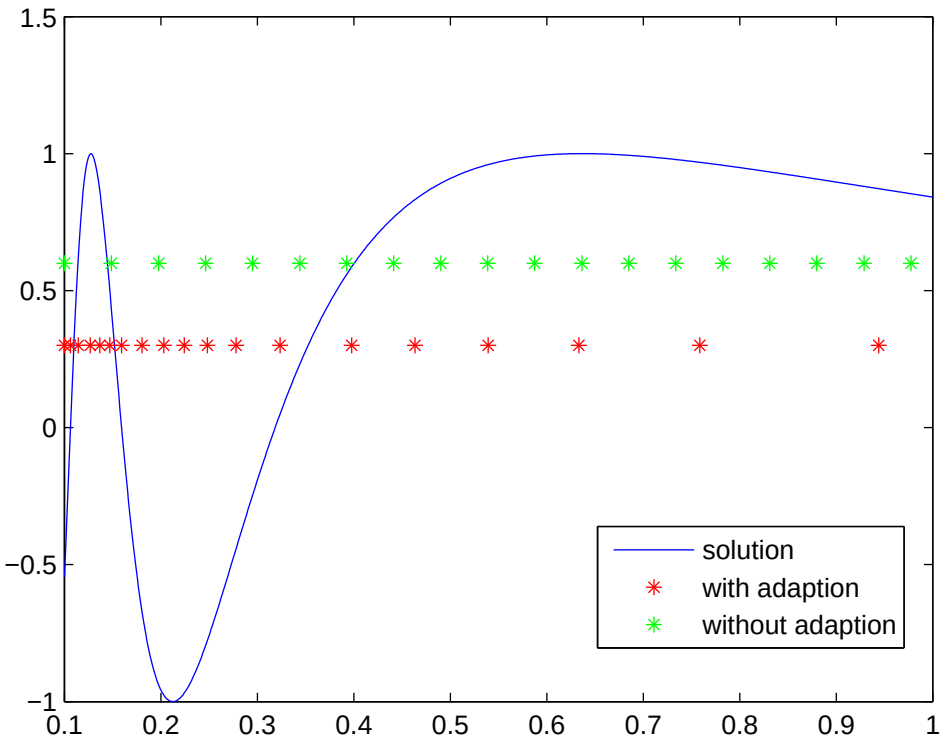

Figure 2: Mesh densities with and without adaptation

As can be seen, the solution has much higher derivatives at the left boundary of the interval than at the rest of the domain. Both the initial solution and the solution grid were chosen with uniformly distributed grid points. In the figure the stars mark the location of each hundredth collocation point.

Without mesh adaptation the collocation grid will stay uniformly distributed. In such an example this leads to a global error which is far larger than the local error on most of the interval, resulting in the need for very fine meshes to reach a given tolerance. As this example shows the mesh adaptation aims to concentrate the collocation points where the solution is less smooth, leading to a roughly evenly distributed error on each collocation interval.

-
```
    case 'errorEstimate'
        ret=0;
```

errorEstimate is another switch taking the values 0 and 1. If it is toggled on, `bvpsuite` will estimate the error by solving the problem first on the requested grid and then on a finer grid with two times the number of points. Using both solutions the error is estimated factoring in the expected order of the solver. Using this option will generate an errest field in the solution structure where the estimated error is stored.

- ```
  case 'absTolSolver'
      ret=1e-12;
  ```

  This is a parameter for the solver solve_`nonlinear_sys`. The routine will terminate if its estimated error is below the bound given by this parameter.

- ```
  case 'relTolSolver'
      ret=1e-12;
  ```

  Similarly the solver will terminate if it finds that its estimated relative error is below this border.

- ```
  case 'absTolMeshAdaptation'
      ret=1e-9;
  case 'relTolMeshAdaptation'
      ret=1e-9;
  ```

  Similarly as the above branches tell the solver when to accept its solution, these two parameters are used by the mesh adaptation routine - if activated - to determine if it should make another attempt to redistribute the mesh or make it finer or it the currently found mesh is suited well enough.

- ```
  case 'finemesh'
      ret = 0;
  ```

  This switch decides if the mesh adaptation routine - in case it is enabled - uses data of the finer mesh the error estimating routine provides. As both of those routines increase the computation time in the average case, it is unchecked by default. For complicated equations however it may be well worth it to give the adaptation additional data.

- ```
  case 'allowTRM'
      ret=1;
  ```

  As described in Chapter 2, the solver falls back to Matlab routines if both of its own algorithms fail to deliver the expected results. Setting this switch to 0 will prevent this behaviour. While in most cases it is best to use all available resources when tackling a harder problem, during the testing phase the Matlab functions showed
  ```

a tendency to converge against wrong „solutions“ or to continue calculations for thousands of steps without improving the result.

Should such unwanted be observed, a possible way out is to uncheck this switch, causing the solving routine to return with the values it had calculated before trying to switch to the backup Matlab routines.

- ```
  case 'maxFunEvalsTRM'
      ret=90000000;
  case 'maxIterationsTRM'
      ret=90000000;
  ```

  These two settings provide another way to prevent an infinite cycle of the Matlab routines. They force them to abandon their calculations if the given number of iterations or the given number of calls to the bvp file is exceeded.

  The default values allow the routines to run without limit. As they document their progress on output, the user may use that information to decide on when to terminate them using the settings above.

- ```
  case 'lambdaMin'
      ret = 0.001;
  ```

  This value influences the decision making of the nonlinear system solver. Generally speaking, the solver tries to bring a certain value to zero, so if this value is decreasing it can be seen as an improvement. More precisely, the solver accepts a step to be successful if its norm is smaller than $1-\lambda_{min}/2$ times the norm of the previous step.

  Since this and similar formulas affect which of the three algorithms the solver chooses next, changing this parameter will change its behaviour. A larger `lambdaMin` will cause the solver to prefer the predictor-corrector method or even the backup Matlab routines, while a smaller value may prevent this.

- ```
  case 'maxAdaptations'
      ret=18;
  ```

  This parameter concerns the mesh adaptation routine. Since a reworking of the entire mesh is computationally expensive, there is the need for a maximum value of such operations. Modifying this value may help if the mesh adaptation does not terminate in reasonable time.

- ```
  case 'switchToFFNFactor'
      ret=0.5;
  ```

When the solver has entered the predictor-corrector search, it uses this parameter to decide if it is save to switch back to the fast frozen newton routine. More precisely, this change of algorithms is performed if the solver encounters a step with a norm smaller than this factor times the norm of the preceding step.

- ```
  case 'updateJacFactor'
      ret=0.5;
  ```

  As already explained in Chapter 2, the fast frozen newton solver aims to save computation time by reusing a Jacobian matrix for as many steps as feasible. To quantify this goal, the routine is programmed to switch the old matrix for the current one if it happens about a step with a norm larger than this factor times the norm of the previous one. Changing this value may therefore reduce computation time at the risk of letting the solver run into a dead end.

- ```
  case 'K'
      ret=200;
  ```

  This last parameter again concerns the mesh adaptation routine. As stated above, this function tries to concentrate the collocation points in regions where the solution behaves more erratically. In order not to let this optimization get out of hand and prevent changes in the density of several orders of magnitude, the parameter K gives the maximum proportion between any two collocation intervals. Reducing it will lead to a more homogeneous grid while it may be necessary to raise this limit for problems with locally highly unsmooth solutions.

3.3 A Linear Example

In this section we demonstrate the use of `bvpsuite` by some simple examples. In order to avoid distractions by equation details we will choose the first problem to be the simplest model problem imaginable: $y' = y,\ y(0) = 1$ on the interval $[0, 1]$. The solution is $y(t) = e^t$

The following listing shows the complete bvp file and is ready to be used in the given form.

```
function [ret] = ex_linear(request,z,za,zb,zc,t,p,lambda)
switch request
    case 'n'
        ret=1;
    case 'orders'
        ret=[1];
    case 'problem'
        ret=[z(1,2)-z(1,1)];
    case 'jacobian'
        %DON'T CHANGE THIS LINE:
```

```
        ret = zeros(length(feval(mfilename,'problem', ...
            zeros(feval(mfilename,'n'), ...
            max(feval(mfilename,'orders'))+1), [], [], [], 0, ...
            zeros(feval(mfilename,'parameters'),1),0)),feval(mfilename,'n'), ...
            max(feval(mfilename,'orders'))+1);
         ret(1,1,1)=-1;
         ret(1,1,2)=1;
    case 'interval'
        ret = [0,1];
    case 'linear'
        ret=1;
    case 'parameters'
        ret=0;
    case 'c'
        ret = [];
    case 'BV'
        ret=[za(1,1)-1];
    case 'dBV'
        %DON'T CHANGE THIS LINE:
        ret = zeros(max(length(feval(mfilename,'c')), ...
            2-length(feval(mfilename,'c'))), ...
            length(feval(mfilename,'problem',zeros(feval(mfilename,'n'), ...
            max(feval(mfilename,'orders'))+1), [], [], [], 0, ...
            zeros(feval(mfilename,'parameters'),1),0)), ...
            feval(mfilename,'n'), max(feval(mfilename,'orders')));
         ret(1,1,1,1) = 1;
    case 'dP'
        ret=[0];
    case 'dP_BV'
        ret=[0;0;0];
    case 'EVP'
        ret=0;
end
```

We will now, just like before, take a closer list of each branch of this main switch statement, explaining the input and also giving an outlook, how different kinds of problems would have to implemented.

```
function [ret] = ex_linear(request,z,za,zb,zc,t,p,lambda)
```

Beginning with the header, the bvp file function again takes a request string with determines which branch of data is to be read. Unlike the settings file however, calls to this file during the computation also include a list of problem specific information.

The variable `z` corresponds to the function values we called `y` in our equation. It comes as a matrix with the rows corresponding to the index of the unknown function (in case a system of equations is to be solved) and columns corresponding to the order of derivative. Details of handling and accessing it will be given in the sections for the branches 'problem' and 'jacobian'.

The next three variables carry boundary data. `za` contains the function values on the left boundary, while `zb` is located on the right boundary. There is also the option to pose

function values on different points within the interval. This case is covered by `zc`. A more detailed description will be given in the 'BV' and 'dBV' section.

The variable `t`, just as in our equation, corresponds to the input variable which lives on the posed interval. `p` is the vector of unknown parameters, and `lambda` corresponds to the eigenvalue, if the problem to be solved is an eigenvalue problem.

Note that it is not necessary for all those input parameters to actually appear in the file. During the computation all calls to the bvp file are nested in an interface which will hand over only the parameters needed for the problem at hand. In order to be consistent with every kind of possible equation it is however necessary to specify all variables in the header.

- ```
  case 'n'
      ret=1;
  ```

  The variable n corresponds to the number of equations to be solved. Since in our case there is only one equation, we set it to 1. If for example we would like to solve the classical sin-cosine-coupling $y_1' = y_2,\ y_2' = -y_1$, n would be 2.

- ```
  case 'orders'
      ret=[1];
  ```

  This branch gives a vector containing the orders of the given system, i.e. the order of the highest derivative appearing in the individual equations. In our case we have one equation of order one, so this vector contains only a 1. In the case of the sin-cosine-coupling above the vector should read $[1, 1]$. The notation already suggests that varying orders are admitted.

- ```
  case 'problem'
      ret=[z(1,2)-z(1,1)];
  ```

  This is one of the core fields of the bvp file: The representation of the equation(s) itself. First of all it has to be noted that the problem vector is a row vector with each entry containing one equation.

  To understand the content of the one equation we note that `z(i,j)`$:= y_i^{(j-1)}$ using our nomenclature. The equations need to be transformed to have 0 as the right hand side, meaning in our case that the input equation has the form $y' - y = 0$. Since there is only one unknown function, every `i` is set to 1 in the transcription process and every `j` corresponds to the order of derivative plus one.

- ```
  case 'jacobian'
      %DON'T CHANGE THIS LINE:
  ```

```
ret = zeros(length(feval(mfilename,'problem', ...
   zeros(feval(mfilename,'n'), ...
   max(feval(mfilename,'orders'))+1), [], [], [], 0, ...
   zeros(feval(mfilename,'parameters'),1),0)), ...
   feval(mfilename,'n'), max(feval(mfilename,'orders'))+1);
 ret(1,1,1)=-1;
 ret(1,1,2)=1;
```

For its computations the solver needs the derivatives of the equations with respect to the unknown function and its derivatives. These are stored in a three dimensional tensor which is initialized in the first line to have the correct size regardless of the actual derivatives.

When filling in the positions of the return tensor, note that `ret(i,j,k)` corresponds to the `i`-th equation being differentiated with respect to the formal parameter `z(j,k)`. Since there is only one equation, `i` will always be equal to 1.

Our equation contains both `z(1,1)` and `z(1,2)`. Differentiation with respect to `z(1,1)` yields -1, while differentiating with respect to `z(1,2)` gives 1. These two values are written into the return tensor according to the given index rules.

- ```
  case 'interval'
      ret = [0,1];
  ```

  This branch returns the interval the problem is posed on. While the syntax is natural enough, note that it is possible to pose semi infinite intervals for example by `ret = [4,Inf]`.

- ```
  case 'linear'
      ret=1;
  ```

  This flag tells the solver whether the problem is linear. Note that there is not much time lost if it is wrongly set to 0, as the fast frozen newton will converge to the solution in just one step. Setting the flag to 1 if the problem is nonlinear will result in unexpected behaviour and a solution which is wrong if the program terminates at all. Note however, that for classes of problems which cannot be linear (for example eigenvalue problems) this flag is ignored by the interface and internally replaced by a 0.

- ```
  case 'parameters'
      ret=0;
  ```

  The parameters branch returns an integer corresponding to the number of unknown parameters appearing in the problem. Since there are none in our example, we set it to 0.
  ```

- ```
  case 'c'
      ret = [];
  ```

  `c` is an optional vector of points where the user may want to prescribe function values. Per default conditions on the solution can only be posed on the left and the right border of the interval.

- ```
  case 'BV'
      ret=[za(1,1)-1];
  ```

  Like the equations were written into the problem branch, this is where the conditions on the boundaries are defined. The syntax is similar to the one involving the variable `z`: The first index in the brackets corresponds to the number of the unknown function while the second one corresponds to the order of derivative. `za` evaluates the solution at the left border of the interval, while `zb` covers the right border.

  Again the right hand side has to be zero, so our boundary condition reads $y-1=0$. In general the return value is a column vector containing all conditions.

- ```
  case 'dBV'
      %DON'T CHANGE THIS LINE:
      ret = zeros(max(length(feval(mfilename,'c')), ...
         2-length(feval(mfilename,'c'))), ...
         length(feval(mfilename,'problem', ...
         zeros(feval(mfilename,'n'), ...
         max(feval(mfilename,'orders'))+1), [], [], [], 0, ...
         zeros(feval(mfilename,'parameters'),1),0)), ...
         feval(mfilename,'n'), max(feval(mfilename,'orders')));
       ret(1,1,1,1) = 1;
  ```

  Here the derivatives of the border conditions are defined. Together with the list of border coordinates, the return value becomes a four dimensional tensor:

  In the expression `ret(h,i,j,k)` the index `h` tells if the derivation is computed with respect to `za` (`h`=1) or `zb` (`h`=2). If custom coordinates for the conditions are used, `h` represents the index in the vector `c` defined above.

  The index `i` corresponds to number of the condition. Since we only pose one condition, this index will always be 1. The last two indices correspond to the ones in the round brackets. For example the line in our example reads: The first condition differentiated with respect to `za(1,1)` is 1.

- ```
  case 'dP'
      ret=[0];
  ```

This lines gives the derivatives of the equations with respect to the unknown parameters. Since there are none in our example, this branch can be skipped.

- ```
  case 'dP_BV'
      ret=[0];
  ```

  The same goes for this branch, which stores the derivatives of the boundary conditions with respect to the unknown parameters. For both this branch and the last it should be noted, that they are not evaluated if the problem is known to feature no unknown parameters. This means that even if invalid data should be „forgotten“ while copying parts of the file for other problems, it won't affect the program as it will never be accessed.

- ```
  case 'EVP'
      ret=0;
  ```

  This last switch tells the routine if the given problem is an eigenvalue problem. If it were one, additional branches would have to be implemented giving information about derivatives with respect to the eigenvalue. Additional details can be found in the tutorials concerning eigenvalue problems.

Having discussed every detail of the bvp file, we note that there are two ways to call the `bvpsuite` routine:

```
[x,y,s]=bvpsuite(@bvpfilename,@settingsfilename);

[x,y,s]=bvpsuite('bvpfilename','settingsfilename');
```

Both options are equivalent in their outcome. Naturally the involved file and the `bvpsuite` functions need to be either in the currently active folder or added to the path.

Note that the function calls display a Matlab exception with identifier `MATLAB:unassignedOutputs`. This is because we did not specify an initial guess, which is not necessary in the linear case. The print is just a remainder that in absence of a user defined initial guess the program switches to default settings.

Figure 3 shows for our example the estimated function values compared to the exact solution using a grid of 100 points with one collocation point.

In Figure 3 the two lines seem nearly congruent. Indeed looking the the absolute differences in Figure 4 it can be seen that the error has an order of magnitude of about 10^{-11}

3.4 A Nonlinear Example

In this section we are going to implement a slightly harder nonlinear problem and explain how initial solutions are passed to the solving routine.

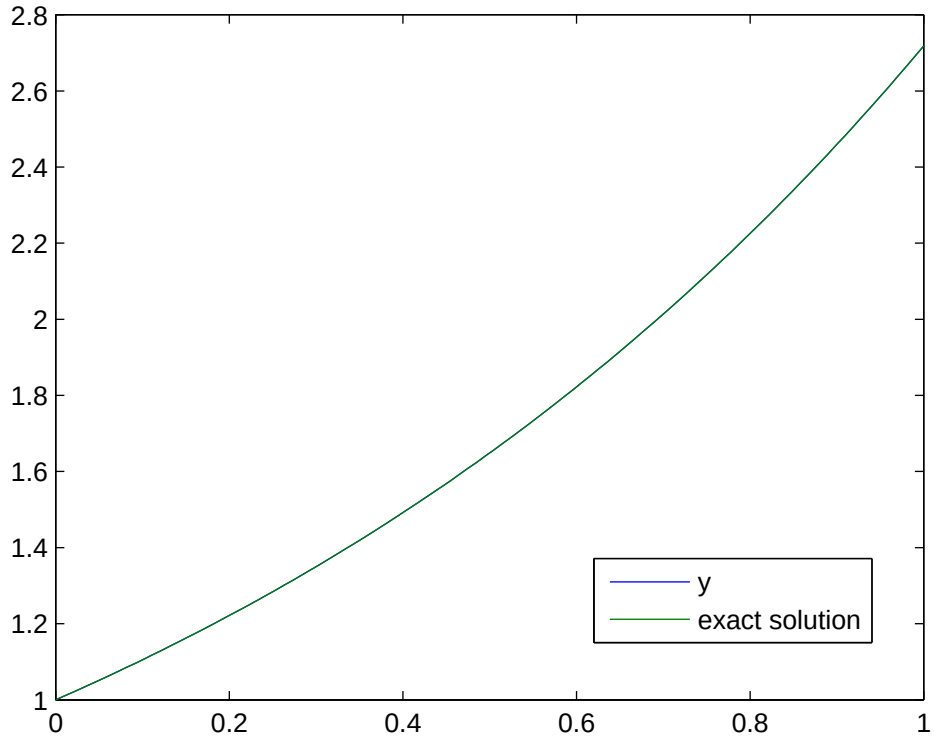

Figure 3: Exact and numerical solution of the linear problem

The equation we are going to consider is

$$y''^2 = 4y^2 \cdot (1 + y^2) \cdot y', \quad y(-1) = \tan(-1), \ \frac{y(1)}{y(-1)} = -1.$$

The exact solution is given by $y(t) = \tan(t)$.

The following listing implements the equation above and can be plugged in just the way it is.

```
function [ret] = ex_nonlin2(request,z,za,zb,zc,t,p,lambda)
switch request
    case 'n'
        ret=1;
    case 'orders'
        ret=[2];
    case 'problem'
        ret=[z(1,3)^2-4*z(1,1)^2*(1+z(1,1)^2)*z(1,2)];
    case 'jacobian'
        %DON'T CHANGE THIS LINE:
        ret = zeros(length(feval(mfilename, 'problem', ...
            zeros(feval(mfilename,'n'), ...
```

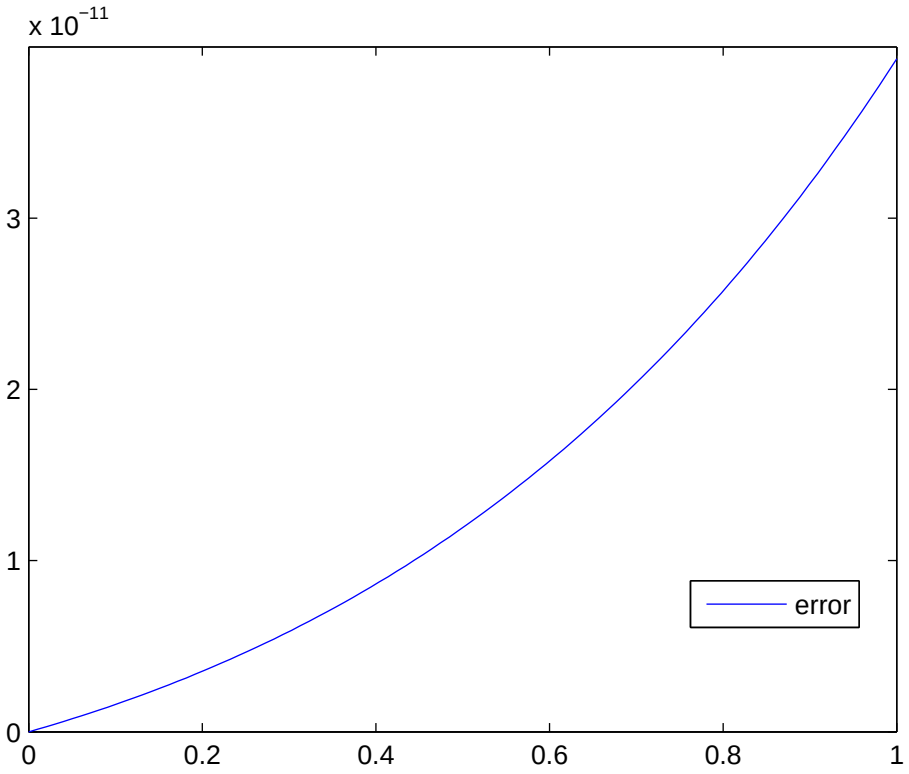

Figure 4: Error of the linear problem

```
        max(feval(mfilename,'orders'))+1),[],[],[], ...
        0,zeros(feval(mfilename,'parameters'),1),0)), ...
        feval(mfilename,'n'), max(feval(mfilename,'orders'))+1);
     ret(1,1,1)=-8*z(1,1)*(1+z(1,1)^2)*z(1,2)-8*z(1,1)^3*z(1,2);
     ret(1,1,2)=-4*z(1,1)^2*(1+z(1,1)^2);
     ret(1,1,3)=2*z(1,3);
case 'interval'
    ret = [-1,1];
case 'linear'
    ret=0;
case 'parameters'
    ret=0;
case 'c'
    ret = [];
case 'BV'
    ret=[za(1,1)-tan(-1);
         zb(1,1)/za(1,1)+1];
case 'dBV'
    %DON'T CHANGE THIS LINE:
    ret = zeros(max(length(feval(mfilename,'c')), ...
       2-length(feval(mfilename,'c'))), length(feval(mfilename, ...
       'problem', zeros(feval(mfilename,'n'), max(feval(mfilename, ...
       'orders'))+1), [],[],[],0, ...
       zeros(feval(mfilename,'parameters'),1),0)), ...
       feval(mfilename,'n'),max(feval(mfilename,'orders')));
```

```
        ret(1,1,1,1)=1;
        ret(1,2,1,1)=-zb(1,1)/za(1,1)^2;
        ret(2,2,1,1)=1/za(1,1);
    case 'dP'
        ret=[0];
    case 'dP_BV'
        ret=[0;0;0];
    case 'initProfile'
        ret.initialMesh = linspace(-1,1,50);
        ret.initialValues = (ret.initialMesh).^3;
    case 'EVP'
        ret=0;
end
```

In the following we look at a few key sections highlighting the differences to the previous example.

```
    case 'problem'
        ret=[z(1,3)^2-4*z(1,1)^2*(1+z(1,1)^2)*z(1,2)];
    case 'jacobian'
        %DON'T CHANGE THIS LINE:
        ret = zeros(length(feval(mfilename, 'problem', ...
            zeros(feval(mfilename,'n'), max(feval(mfilename, ...
            'orders'))+1),[],[],[], 0, zeros(feval(mfilename, ...
            'parameters'),1),0)), feval(mfilename,'n'), ...
            max(feval(mfilename,'orders'))+1);
         ret(1,1,1)=-8*z(1,1)*(1+z(1,1)^2)*z(1,2)-8*z(1,1)^3*z(1,2);
         ret(1,1,2)=-4*z(1,1)^2*(1+z(1,1)^2);
         ret(1,1,3)=2*z(1,3);
```

The problem branch shows our equation translated by the rule `z(i,k)`$= y_i^{(k)}$. The parallels between this line and the target equations should require no further explainations.

The jacobian section is where matters tend to become a bit complicated and error prone with growing problem complexity. It is therefore advisable to use a program like Maple to compute the derivatives.

Our equation differentiated with respect to the formal parameter y gives $-8y \cdot (1+y^2) * y' - 8y^3y'$. Differentiating with respect to y' gives $-4y^2 \cdot (1+y^2)$ and the derivative with respect to y'' has the form $2y''$. These computations are transcribed in the lines above.

```
    case 'BV'
        ret=[za(1,1)-tan(-1);
             zb(1,1)/za(1,1)+1];
    case 'dBV'
        %DON'T CHANGE THIS LINE:
        ret = zeros(max(length(feval(mfilename,'c')), ...
            2-length(feval(mfilename,'c'))), length(feval(mfilename, ...
            'problem', zeros(feval(mfilename,'n'), ...
            max(feval(mfilename,'orders'))+1), [],[],[],0, ...
            zeros(feval(mfilename,'parameters'),1),0)), ...
            feval(mfilename,'n'),max(feval(mfilename,'orders')));
```

```
        ret(1,1,1,1)=1;
        ret(1,2,1,1)=-zb(1,1)/za(1,1)^2;
        ret(2,2,1,1)=1/za(1,1);
```

The `BV` branch should be understandable with the explainations given in the linear example. One only has to remember that za corresponds to the left boundary of the interval and zb to the right one.

In the `dBV` branch the first line states that the first boundary condition differentiated with respect to $za(1,1) = y(-1)$ is 1. In the second one is written that the second condition differentiated with respect to $za(1,1) = y(-1)$ yields $-y(1)/y(-1)^2 = -zb(1,1)/za(1,1)^2$. The final derivative with respect to $zb(1,1) = y(1)$ is given by $1/y(-1) = 1/za(1,1)$.

```
case 'initProfile'
    ret.initialMesh = linspace(-1,1,50);
    ret.initialValues = (ret.initialMesh).^3;
```

This is where the initial profile is defined. The program expects a structure with - in the case of a problem, without parameters or eigenvalues - two fields. The `initalMesh` field gives the grid the initial guess is defined one with `initialValues` providing the corresponding function values.

For our problem we will choose the cubic monomial x^3 evaluated at 50 uniformly distributed points. Figure 5 shows this initial guess in comparison the solution.
It can be seen that our initial guess, while sharing same basic properties with the solution, is by no means a good approximation and it does not satisfy the given boundary conditions.

The final calculation was done like in the linear case using 100 grid points for the solution and two Gaussian collation points in each collocation interval. Since the calculated solution is again not distinguishable from the exact one with the naked eye, we will skip the comparison and move straight to the error plot in Figure 6.

As can be seen, even with the rather crude initial guess and the relatively small number of grid points the error stays globally below $10^{-8)}$. On the borders the error is equal to zero by enforcement of the border conditions. The fact that it vanishes at 0 is most likely due to symmetries of the example. This also shows that a fitting choice for an initial guess helps drastically to improve the precision of the result.

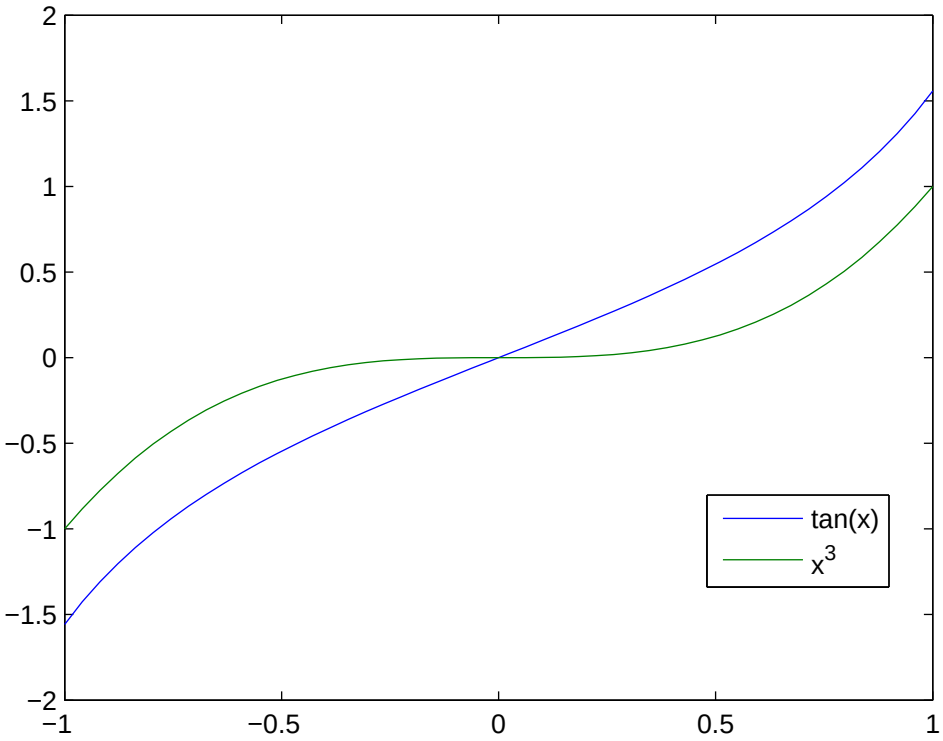

Figure 5: Exact solution and initial guess

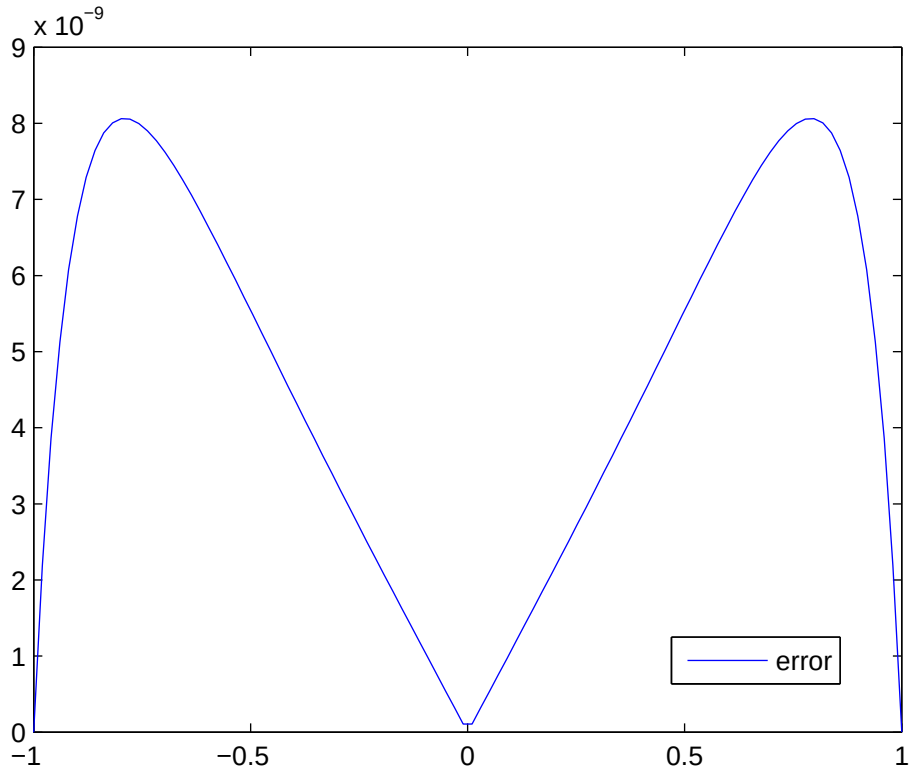

Figure 6: Error of the nonlinear example

4 Application to a degenerate m-Laplacian problem

4.1 Problem Setting

In this chapter we aim to solve a more complicated family of differential equations, featuring versions of a degenerate m-Laplacian. More precisely we consider problems of the form

$$\begin{aligned}&(|y'(x)|^{m-2}y'(x))' + \frac{N-1}{x}|y'(x)|^{m-2}y'(x) + f(y(x)) = 0, \quad 0 < x < M,\\ &y'(0) = y'(M) = y(M) = 0.\end{aligned}$$

The function f is of the form $f(y(x)) = ay^q(x) - by^p(x)$. In the equation setting m, N, a, b, p and q are given parameters. It should be specially noted that the length of the interval M is not given, but appears as an unknown parameter. Hence we need three boundary conditions in order for the problem to be well defined.

In order to arrive at a formulation which is compatible to our code, we evaluate the derivative and normalize the resulting equation to isolate the second derivative:

$$\begin{aligned}&y''(x) + \frac{1}{m-1}\frac{N-1}{x}y'(x) + \frac{1}{m-1}\frac{a\,y^q(x) - b\,y^p(x)}{|y'(x)|^{m-2}} = 0, \quad 0 < x < M,\\ &y'(0) = y'(M) = y(M) = 0.\end{aligned}$$

Since we are interested in solving equations on a fixed interval, we transform the interval by substituting $z := \frac{x}{M}$. The new variable z is defined on the fixed interval $[0, 1]$. Expressing the equation in this new variable and again isolating the second derivative yields

$$\begin{aligned}&y''(z) + \frac{1}{m-1}\frac{N-1}{z}y'(z) + \frac{\lambda}{m-1}\frac{a\,y^q(z) - b\,y^p(z)}{|y'(z)|^{m-2}} = 0, \quad 0 < z < 1,\\ &y'(0) = y'(1) = y(1) = 0.\end{aligned}$$

with $\lambda = M^m$. This way the unknown interval length M appears naturally as an parameter in our equation.

While this formulation is already fit to be plugged into `bvpsuite`, the equation is still problematic, since depending on the parameters m and p, the solution may be singular at one or both endpoints (see [4] for details).

In [3] one can find a list of possible smoothing transformations, depending on the number and location of these singularities. For simplicity we will only discuss the most general one, which can be applied in every case. In the examples below we will however use simpler transformation formulas in case that they are available.

The most general transformation reads

$$t := \left(1 - (1-z)^{\frac{k_2}{2}}\right)^{\frac{k_1}{2}}.$$

Here the two appearing parameters are chosen as $k_1 := \frac{m}{m-1}$ and $k_2 := \frac{m}{m-1-p}$. Applying this transformation to the original equation yields

$$\begin{aligned}
&a_1(t)|y'(t)|^{m-2}[b_1(t)y''(t) + c_1(t)y'(t)] + \lambda(ay(t)^q - by(t)^q) = 0 \quad 0 < t < 1,\\
&y'(0) = y(1) = y'(1) = 0,\\
&\text{where}\\
&a_1(t) := (m-1)\left(\frac{k_1k_2}{4}\right)^{m-1} t^{\left(1-\frac{2}{k_1}\right)(m-1)} \left(1 - t^{\frac{2}{k_1}}\right)^{\left(1-\frac{2}{k_1}\right)(m-1)},\\
&b_1(t) := \frac{k_1k_2}{4} t^{1-\frac{2}{k_1}} \left(1 - t^{\frac{2}{k_1}}\right)^{1-\frac{2}{k_2}},\\
&c_1(t) := \frac{1}{4} t^{-\frac{2}{k_1}} \left(1 - t^{\frac{2}{k_1}}\right)^{-\frac{2}{k_2}} \left[4 - 2k_2 + \left(1 - t^{\frac{2}{k_1}}\right)(-4 + k_1k_2)\right]\\
&\qquad + \frac{N-1}{(m-1)\left(1 - \left(1 - t^{\frac{2}{k_1}}\right)^{\frac{2}{k_2}}\right)}.
\end{aligned}$$

In the following we will consider four special choices of parameters.

4.2 Case 1

First we consider the set of parameters given as $p = -\frac{1}{2}$, $q = 1$, $a = 1$, $b = 1$, $m = \frac{3}{2}$, $N = 3$. This choice leads to the equation

$$\begin{aligned}
&y''(z) + \frac{4}{z} y'(z) + 2\lambda \left(y(z) - \frac{1}{\sqrt{y(z)}}\right) \sqrt{|y'(z)|} = 0,\\
&y'(0) = y'(1) = y(1) = 0.
\end{aligned}$$

The code was unable to find a solution for this equation.

Applying the smoothing transformation $t = 1 - (1-z)^{\frac{3}{4}}$ we obtain the modified equation

$$\begin{aligned}
&\frac{3\sqrt{3}}{16\sqrt{1-t}} y''(z) + \left(\frac{\sqrt{3}}{16(1-t)^{\frac{3}{2}}} + \frac{\sqrt{3}}{(1-t)^{\frac{1}{6}}\left(1 - (1-t)^{\frac{4}{3}}\right)}\right) y'(z)\\
&+ \lambda\sqrt{|y'(z)|}\left(y(z) - \frac{1}{\sqrt{y(z)}}\right) = 0,\\
&y'(0) = y'(1) = y(1) = 0.
\end{aligned}$$

Using one Gaussian point per collocation interval the solver did converge. The numerical solution is visualized in Figure 7.

A quantitative overview of the converging progress is given in Table 1. Here and in the rest of the chapter the column „intervals“ gives the number of collocation intervals used in the specific step. The reason for the slightly odd numbers is the fact that a distribution of the whole interval by for example one hundred points leads to one hundred and one intervals in total.

The column „error“ reads the maximum of the measured errors in respect to the exact solution, calculated over all collocation points. In cases like this one where the exact solution is not known the numerical solution with 12801 intervals is used as a reference solution. The reference solution was calculated using one collocation point.

The column „error λ“ gives the absolute difference of the numerically calculated parameter λ to the exact one. Again the numerical solution using 12801 intervals and one collocation point is used if the exact value of λ is not known.

The two columns „rate“ and „rate λ“ show the numerical convergence rate from the previous step to the current one (therefore the first line will always be empty). Note that the expected theoretical rates are two for one Gaussian point, four for two and six for three points.

As we can see, the results are as expected with rates consistently near two. Since the solver did not converge for any other number of collocation points however, we cannot check the other outcomes.

A visualization of the convergence can be found in Figure 8.

4.3 Case 2

Next we consider the parameters $p = \frac{1}{2}$, $q = 1$, $a = 1$, $b = 1$, $m = 3$, $N = 3$. Thus the equation to consider is

$$y''(z) + \frac{1}{z}\,y'(z) + \frac{\lambda}{2}\,\frac{y(z) - \sqrt{y(z)}}{|y'(z)|} = 0, \quad y'(0) = y'(1) = y(1) = 0.$$

This choice has the benefit that the solution is known to be $y(z) = \left(2 - 2z^{\frac{3}{2}}\right)^2$ with the parameter λ being equal to 216. This means that all data can be compared with the exact values.

Tables 2 and 3 show the numerical results using one and two Gaussian points, respectively.

Figure 9 gives a more graphical presentation of these results.

While the solver did converge with stable rates, these rates fall behind the theoretical optimal rates by a factor of about two, with the exception of the parameter with one point.

The reason for this can be attributed to the fact that the problem is singular, which is known to cause order reductions. Another run with three Gaussian points did not converge.

While the code is apparently able to handle singular problems, this also shows why it may still be worth putting some work into smoothing transformations, in case one attempts to achieve high precision of say 1e-10. This would require a tremendous amount of intervals and therefore CPU-time (assuming the convergence rate will stay stable). We will see later that such precision can be easily achieved after a smoothing step.

The chosen smoothing transformation in this case reads $t = z^{\frac{3}{4}}$. Applying it to the original equation yields

$$\begin{aligned} &\frac{27}{32t}y''(t) + \frac{27}{32t^2}\,y'(t) + \lambda\,\frac{y(t) - \sqrt{y(t)}}{|y'(t)|} = 0, \\ &y'(0) = y'(1) = y(1) = 0. \end{aligned}$$

The corresponding transformed exact solution is $y(t) = (2 - 2t^2)^2$ with λ still being 216. Figure 10 shows the original and transformed solution of the problem.

Tables 4, 5 and 6 report on numerical results using one, two and three Gaussian points.

As can be seen, the versions with one and two Gaussian points work quite well. The convergence rate still does not completely live up to the expectations, but the two point version still manages to reach a precision of 1e-10 with the given amount of intervals.

The rates in Table 6 are of course without meaning. Even with just fifty intervals the numerical solution is already an order of magnitude above calculating accuracy, meaning that no further improvement can be reasonably expected. Further tests going as low as three intervals showed the same behaviour however, so it is impossible to expect a meaningful rate of convergence. The results are visualized in Figure 11.

4.4 Case 3

Here we consider the parameters $p = -\frac{1}{2}$, $q = 1$, $a = 1$, $b = \frac{2}{3}$, $m = 2$, $N = 1$. Now the original equation reads

$$\begin{aligned} &y''(z) + \lambda\left(y(z) - \frac{2}{3\sqrt{y(z)}}\right) = 0, \\ &y'(0) = y'(1) = y(1) = 0. \end{aligned}$$

Again the exact solution is known, this time being $y(z) = \left(\frac{8}{3}\right)^{\frac{2}{3}} \left(\cos\left(\frac{\pi}{2}z\right)\right)^{\frac{4}{3}}$ with $\lambda = \left(\frac{2\pi}{3}\right)^2$.

Just as in the second case, the solver only converged with up to two Gaussian points. The results can be found in Tables 7 and 8.

While a reduction of order was to be expected, it is a bit surprising how close to 1 the rate is. Apart from that there is nothing unexpected with those results and they show once again that without smoothing high precision is not achievable using efficient numbers of intervals. It also is a perfect illustration that there is not much to gain by using higher order methods for an unsmooth problem.

A visualization can be found in Figure 12.

Applying the smoothing transformation $t = 1 - (1-z)^{\frac{2}{3}}$ the problem takes the form

$$\begin{aligned} &\frac{4}{9(1-t)} y''(t) + \frac{2}{9\,(t-1)^2} y'(t) + \lambda \left(y(t) - \frac{2}{3\sqrt{y(t)}} \right) = 0, \\ &y'(0) = y'(1) = y(1) = 0. \end{aligned}$$

The according solution is given by $y(t) = \left(\frac{8}{3}\right)^{\frac{2}{3}} \left(\cos\left(\frac{\pi}{2}\left(1-(1-t)^{\frac{3}{2}}\right)\right)\right)^{\frac{4}{3}}$, $\lambda = \left(\frac{2\pi}{3}\right)^2$. A graphical presentation of the smooth and unsmooth solution can be seen in Figure 13.

Again we compute the numerical solutions using one, two and three Gaussian points. The results can be found in Tables 9, 10, and 11, respectively.

The tables show nearly perfect rates for the versions with one and two points. Even if it does not quite reach the ideal order of six, the version with three points does come within an order of magnitude to calculating precision (the flat zero in the last line must be seen as a numerical effect).

Figure 14 shows the data in a more compact form.

4.5 Case 4

In this last case we consider the parameters $p = 0$, $q = 1$, $a = 1$, $b = 1$, $m = 3$, $N = 3$. This leads to an original problem of the form

$$\begin{aligned} &y''(z) + \frac{1}{z}\, y'(z) + \frac{\lambda}{2}\, \frac{y(z) - 1}{|y'(z)|} = 0, \\ &y'(0) = y'(1) = y(1) = 0. \end{aligned}$$

Just like in the first case the solver does not converge. We therefore apply the smoothing transformation $t = \left(1 - (1-z)^{\frac{3}{4}}\right)^{\frac{3}{4}}$. This leads to

$$\frac{729}{2048t\left(1-t^{\frac{4}{3}}\right)}y''(z)+\lambda\frac{y(z)-1}{|y'(z)|}$$
$$+\left(\frac{81}{512t^{\frac{2}{3}}\left(1-t^{\frac{4}{3}}\right)^2}-\frac{243}{2048t^2\left(1-t^{\frac{4}{3}}\right)}+\frac{81}{128t^{\frac{2}{3}}\left(1-t^{\frac{4}{3}}\right)^{\frac{2}{3}}\left(1-\left(1-t^{\frac{4}{3}}\right)^{\frac{4}{3}}\right)}\right)y'(z)=0,$$
$$y'(0)=y'(1)=y(1)=0.$$

Using an intricate starting solution the solver converges to the numerical solution which can be seen in Figure 15.

Tables 12, 13 and 14 show consistent, if not perfect rates of convergence.

Since the exact solution is not known, the numerical solution calculated using 12801 intervals was used as reference, calculated with three collocation points. We can see that the calculations using one Gaussian collocation point admit a perfect rate of 2.

While the version with two Gaussian collocation points starts of with errors four orders of magnitude smaller than the one using only one point, the rates of convergence have not improved, or only slightly in the case of the parameter. A third collocation point leads to a very stable rate of roughly 8/3, which, while being far from the optimal rate, is still enough to satisfy standard tolerances.

A visual representation of these results can be found in Figure 16.

4.6 Figures and Tables

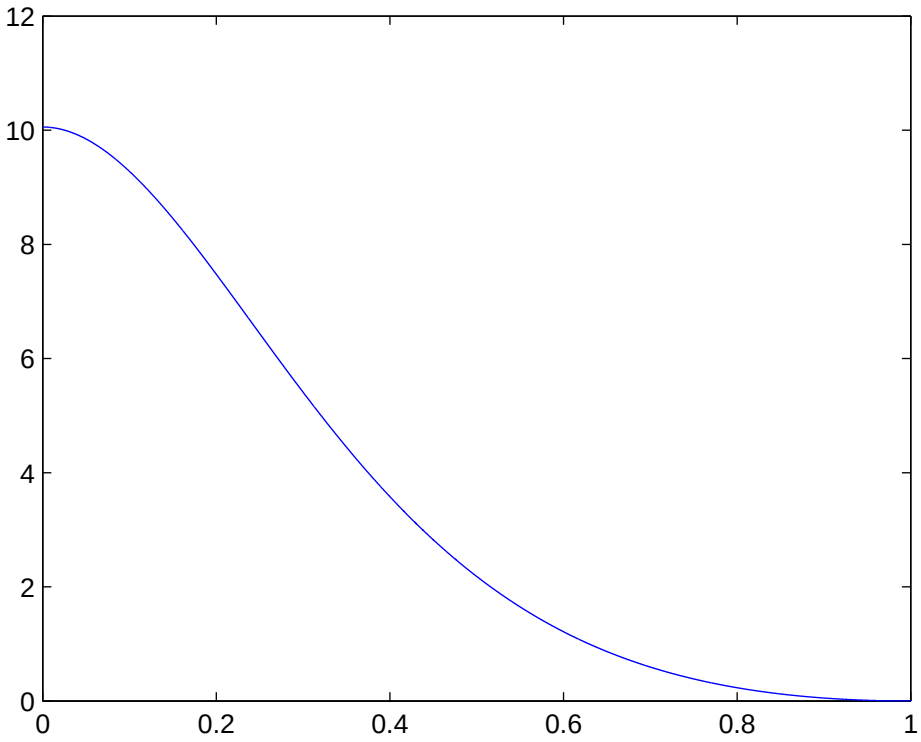

Figure 7: Case 1 smooth: Numerical solution

intervals	**error**	**rate**	**error λ**	**rate λ**
101	1.4159e-3	-	7.7077e-4	-
201	3.7705e-4	1.9089	2.0018e-4	1.9450
401	9.8751e-5	1.9329	5.1658e-5	1.9542
801	2.5368e-5	1.9608	1.3163e-5	1.9725
1601	6.2294e-6	2.0258	3.2199e-6	2.0315

Table 1: Case 1 smooth: Estimated errors with one Gaussian point

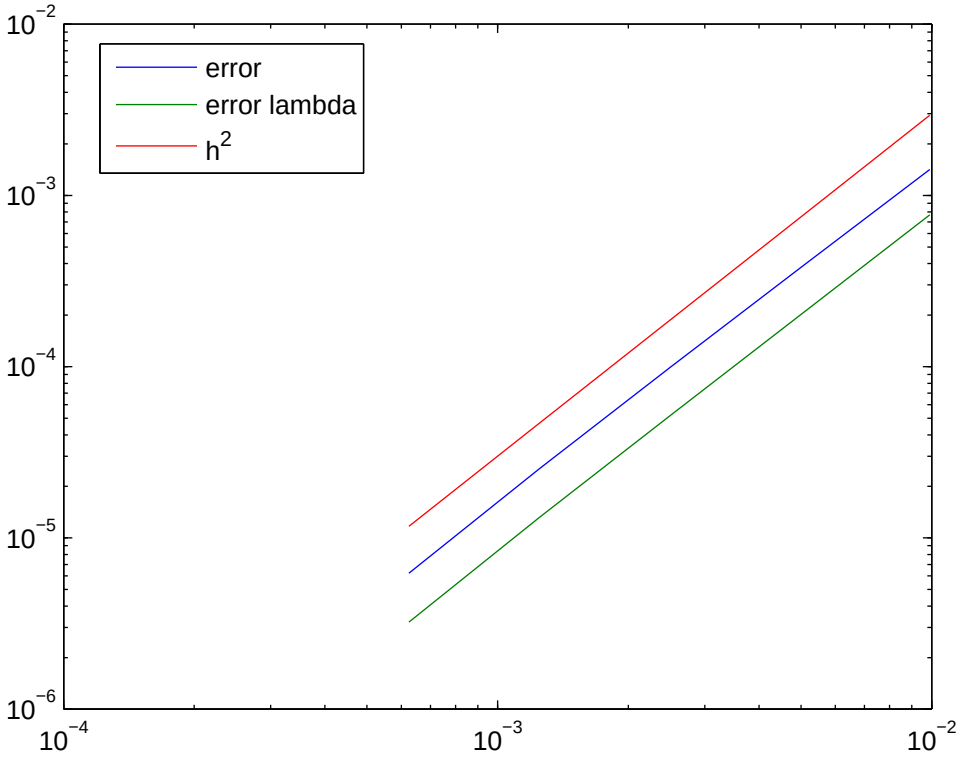

Figure 8: Case 1 smooth: Errors

intervals	**error**	**rate**	**error λ**	**rate λ**
51	5.9704e-3	-	9.0537e-2	-
101	2.1840e-3	1.4508	2.2923e-2	1.9817
201	7.9267e-4	1.4622	5.9503e-3	1.9458
401	2.8585e-4	1.4715	1.5591e-3	1.9323
801	1.0255e-4	1.4789	4.0965e-4	1.9282

Table 2: Case 2 unsmooth: Errors with one Gaussian point

intervals	**error**	**rate**	**error λ**	**rate λ**
51	7.7497e-4	-	1.0312e-3	-
101	2.6717e-4	1.5364	1.2834e-4	3.0063
201	9.3530e-5	1.5143	1.5979e-5	3.0057
401	3.2930e-5	1.5060	1.9922e-6	3.0037
801	1.1621e-5	1.5027	2.4866e-7	3.0021

Table 3: Case 2 unsmooth: Errors with two Gaussian points

intervals	**error**	**rate**	**error λ**	**rate λ**
51	3.2263e-4	-	1.2405e-2	-
101	7.7972e-5	2.0489	6.8935e-3	0.8476
201	1.9219e-5	2.0204	2.6530e-3	1.3776
401	4.9903e-6	1.9453	8.9509e-4	1.5675
801	1.4148e-6	1.8185	2.8184e-4	1.6672

Table 4: Case 2 smooth: Errors with one Gaussian point

intervals	**error**	**rate**	**error λ**	**rate λ**
51	8.4864e-8	-	1.4136e-5	-
101	8.2352e-9	3.3653	1.5283e-6	3.2094
201	8.9200e-10	3.2067	1.7736e-7	3.1071
401	1.0324e-10	3.1110	2.1353e-8	3.0542
801	1.2399e-11	3.0577	2.6192e-9	3.0272

Table 5: Case 2 smooth: Errors with two Gaussian points

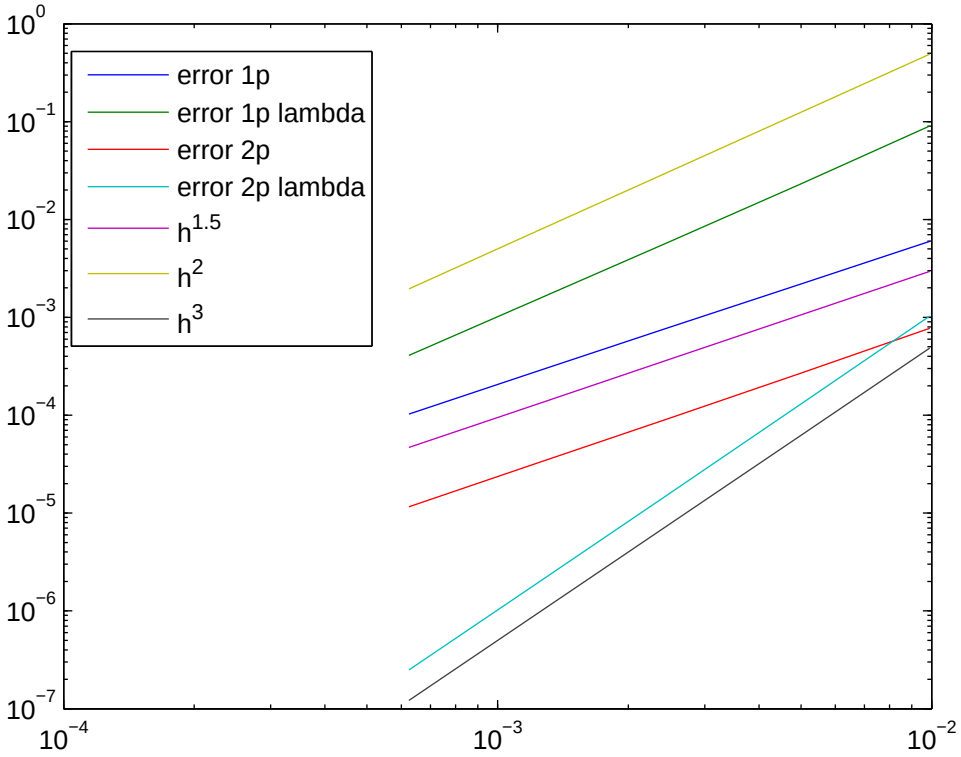

Figure 9: Case 2 unsmooth: Errors

intervals	**error**	**rate**	**error λ**	**rate λ**
51	3.5527e-15	-	8.5265e-14	-
101	1.7764e-15	1.0000	5.1159e-13	-2.5850
201	4.4409e-15	-1.3219	3.1264e-13	0.7105
401	2.2204e-15	1.0000	2.8422e-14	3.4594
801	6.6613e-15	-0.5850	2.5580e-13	-3.1699

Table 6: Case 2 smooth: Errors with three Gaussian points

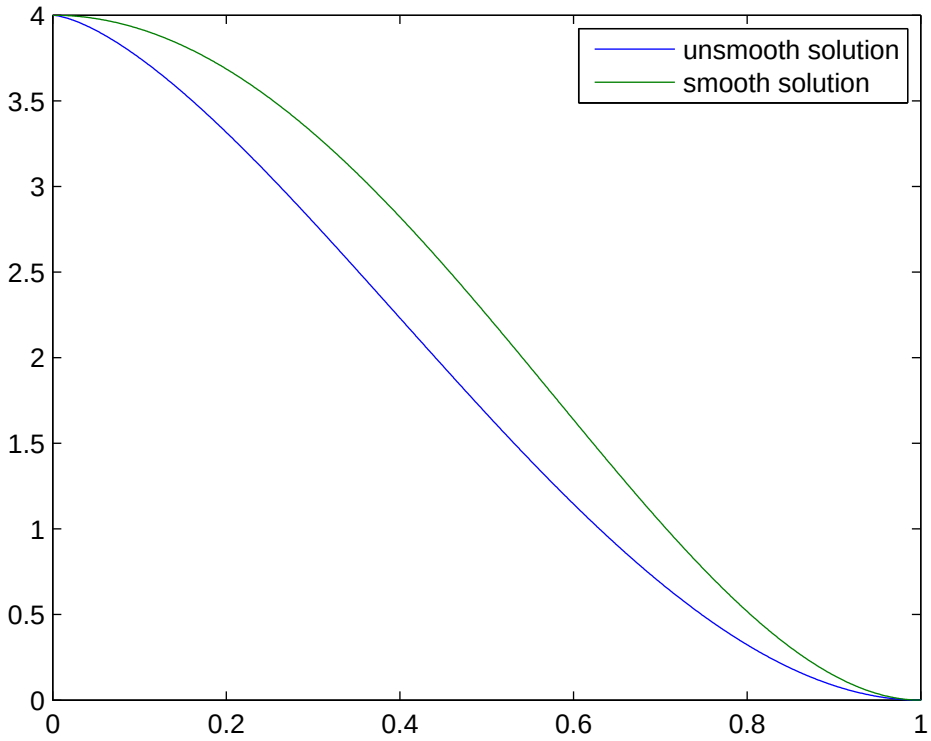

Figure 10: Case 2: Numerical Solutions of the smooth and unsmooth formulation

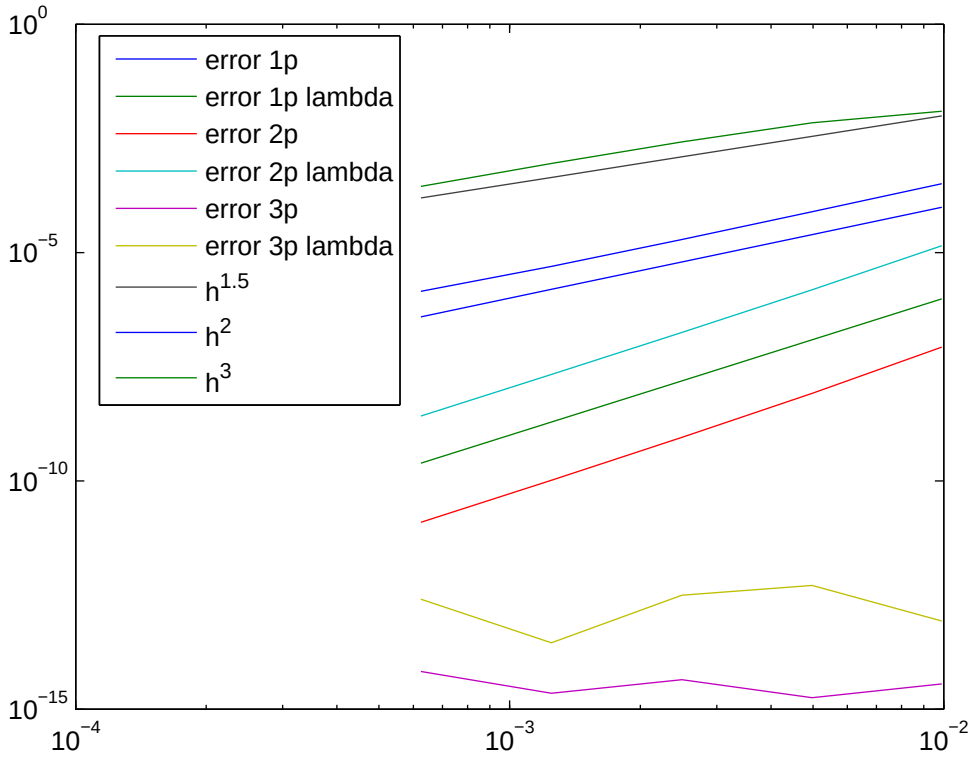

Figure 11: Case 2 smooth: Errors

intervals	**error**	**rate**	**error λ**	**rate λ**
51	1.6264e-2	-	7.2480e-2	-
101	8.4085e-3	0.9517	3.7381e-2	0.9553
201	4.4075e-3	0.9319	1.9636e-2	0.9288
401	2.3402e-3	0.9133	1.0470e-2	0.9072
801	1.2589e-3	0.8944	5.6600e-3	0.8874

Table 7: Case 3 unsmooth: Errors with one Gaussian point

intervals	**error**	**rate**	**error λ**	**rate λ**
51	5.3583e-3	-	2.2775e-2	-
101	2.6550e-3	1.0130	1.1270e-2	1.0150
201	1.3230e-3	1.0049	5.6147e-3	1.0052
401	6.6135e-4	1.0003	2.8075e-3	0.9999
801	3.3126e-4	0.9974	1.4071e-3	0.9966

Table 8: Case 3 unsmooth: Errors with two Gaussian points

intervals	**error**	**rate**	**error λ**	**rate λ**
51	3.6880e-4	-	9.9320e-4	-
101	9.0381e-5	2.0288	2.4342e-4	2.0286
201	2.2371e-5	2.0144	6.0254e-5	2.0143
401	5.5650e-6	2.0072	1.4989e-5	2.0072
801	1.3878e-6	2.0036	3.7379e-6	2.0036

Table 9: Case 3 smooth: Errors with one Gaussian point

intervals	**error**	**rate**	**error λ**	**rate λ**
51	4.2346e-8	-	1.4065e-7	-
101	2.9273e-9	3.8546	1.0205e-8	3.7848
201	2.0278e-10	3.8516	7.3741e-10	3.7907
401	1.3995e-11	3.8569	5.2773e-11	3.8046
801	9.6101e-13	3.8642	3.7295e-12	3.8228

Table 10: Case 3 smooth: Errors with two Gaussian points

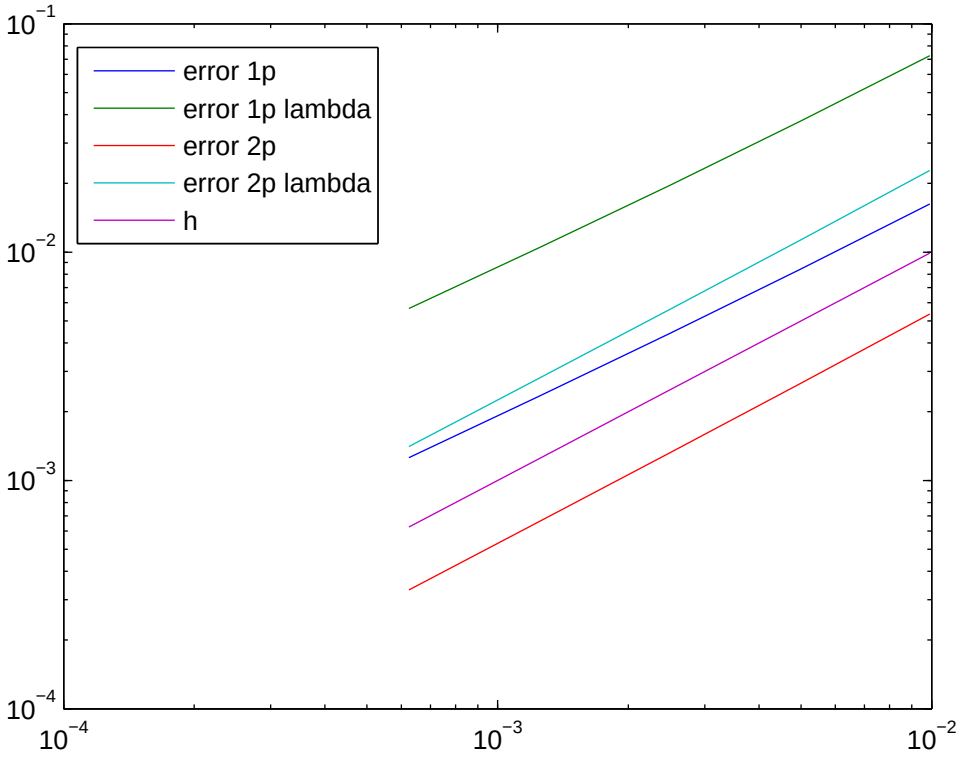

Figure 12: Case 3 unsmooth: Errors

intervals	**error**	**rate**	**error λ**	**rate λ**
51	3.2894e-10	-	1.4718e-9	-
101	1.5624e-11	4.3960	7.0387e-11	4.3862
201	7.8326e-13	4.3182	3.5500e-12	4.3094
401	4.1411e-14	4.2414	1.8385e-13	4.2712
801	2.8866e-15	3.8426	0	∞

Table 11: Case 3 smooth: Errors with three Gaussian points

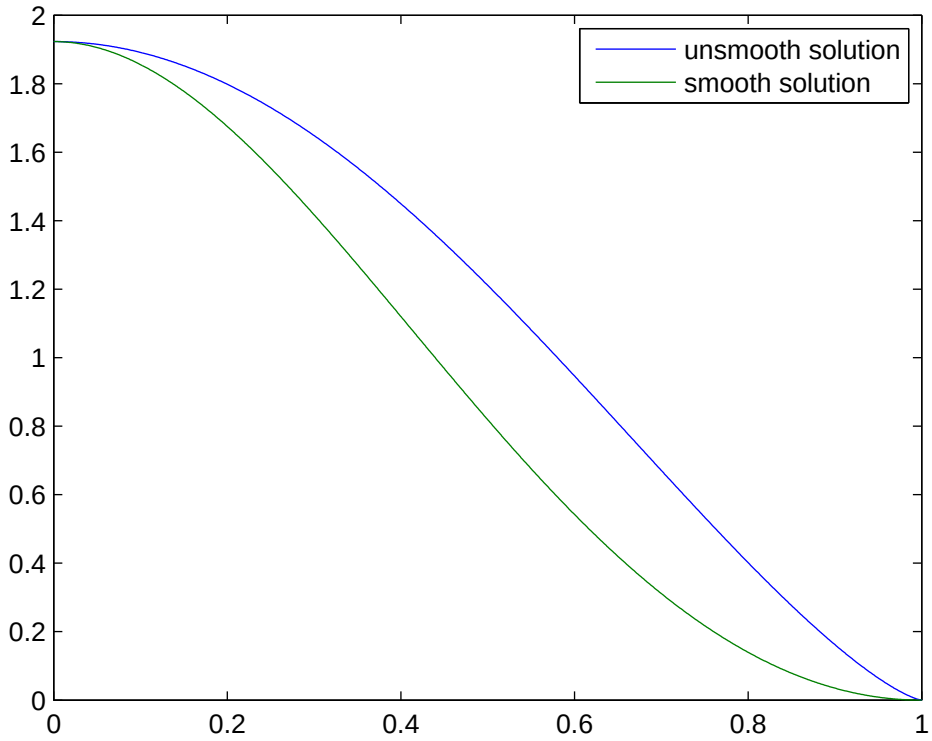

Figure 13: Case 3: Numerical Solutions of the smooth and unsmooth formulation

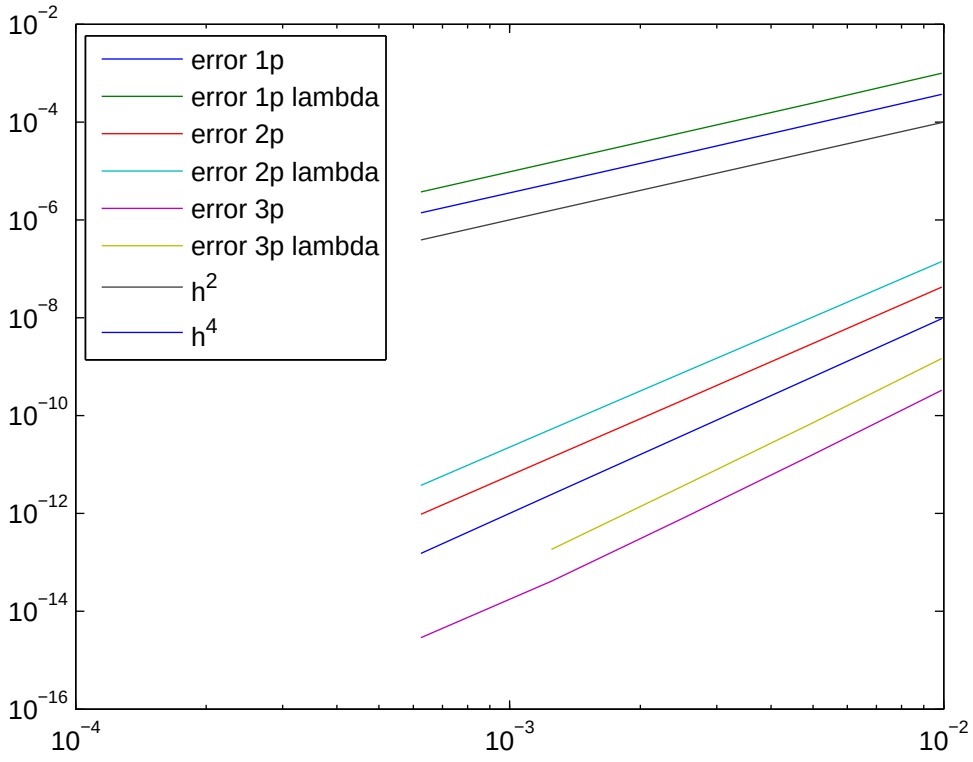

Figure 14: Case 3 smooth: Errors

intervals	**error**	**rate**	**error λ**	**rate λ**
101	2.9690e-4	-	1.0392e-2	-
201	7.4201e-5	2.0005	2.6481e-3	1.9724
401	1.8546e-5	2.0003	6.7276e-4	1.9768
801	4.6362e-6	2.0001	1.7044e-4	1.9808
1601	1.1590e-6	2.0001	4.3074e-5	1.9844

Table 12: Case 4 smooth: Errors with one Gaussian point

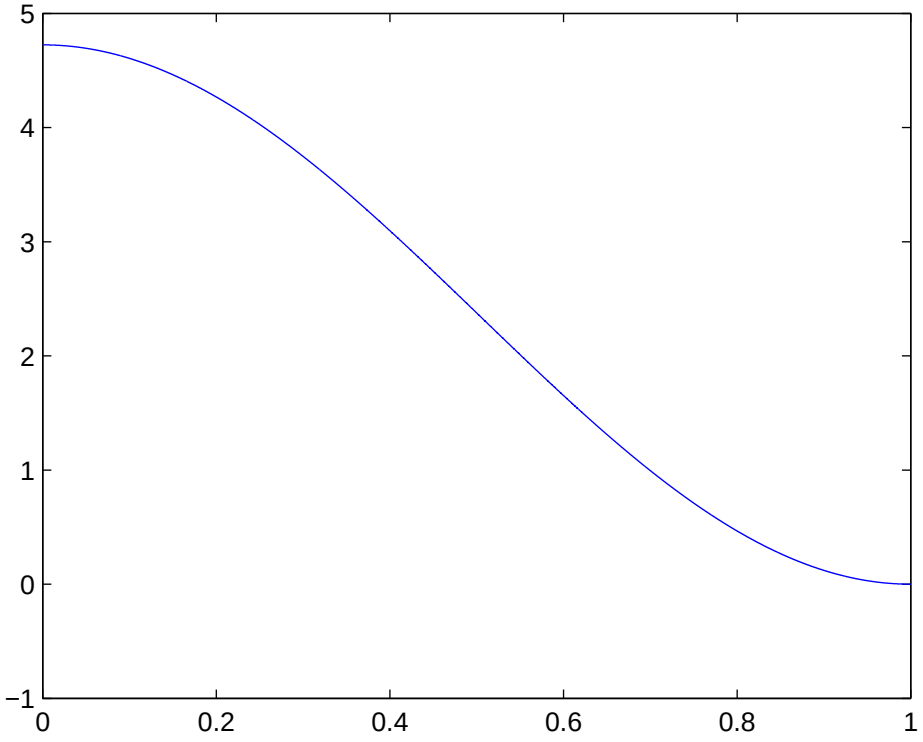

Figure 15: Case 4 smooth: Numerical solution

intervals	error	rate	error λ	rate λ
101	5.4169e-8	-	4.094e-6	-
201	1.4366e-8	2.0005	1.1438e-6	1.8398
401	2.8523e-9	2.0003	2.3390e-7	2.2898
801	5.0821e-10	2.0001	4.2424e-8	2.4629
1601	8.5791e-11	2.0001	7.2407e-9	2.5507

Table 13: Case 4 smooth: Errors with two Gaussian points

intervals	error	rate	error λ	rate λ
101	1.1207e-8	-	8.8462e-7	-
201	1.7684e-9	2.6639	1.4439e-7	2.6151
401	2.7833e-10	2.6676	2.3178e-8	2.6391
801	4.3667e-11	2.6722	3.6802e-9	2.6549
1601	6.7282e-12	2.6983	5.7184e-10	2.6861

Table 14: Case 4 smooth: Errors with three Gaussian points

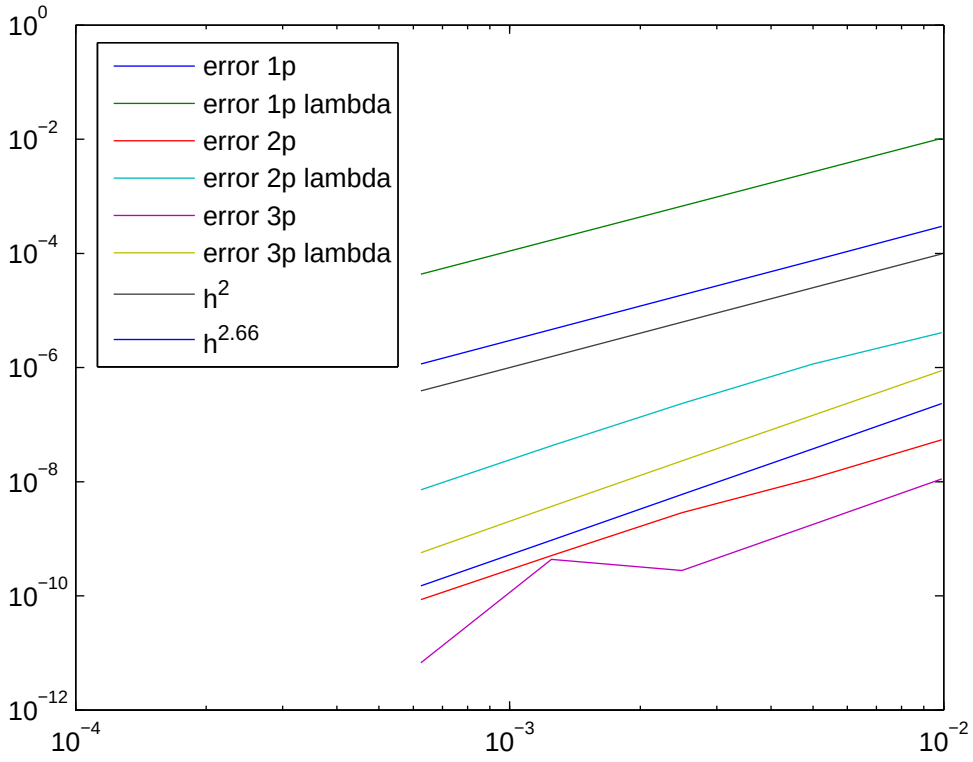

Figure 16: Case 4 smooth: Errors

5 Conclusions

In the course of this thesis we have been talking about principles of collocation applied to boundary value problems. We also covered variants of the Newton method to numerically solve the algebraic system involved in applying the collocation method to nonlinear problems. We then had a look at the routine `solve_nonlinear_sys`, which is based on these techniques and was largely unaltered in the refactoring process.

In Chapter 3 we discussed some of the differences between the old version of `bvpsuite` and the new one and gave a manual-like introduction to its basic handling. The most obvious change to the user was the separation of the problem data into the bvp file containing information about the equations and additional conditions and a settings file which can be used independently.

On code level a big improvement was the refactoring of the large and unintuitive Matlab functions of the old version - especially `run.m` and `equations.m` - into smaller routines with clear interfaces. As an example the newly created `SolveNonlinearProblem` was discussed in more detail.

Chapter 4 showed an application of the code to a couple of more advanced problems. The code performed well overall with overall decent, if not optimal rates of convergence. In the cases 1 and 4 the solver did not converge using the unsmooth formulation, however in both cases the smooth formulation was solvable.

Largely the new version performed very much like the old one, since there have been only few changes to the underlying solver routines. However the old version did have one unusual quirk in case 4 using two Gaussian points. While the error of λ converged against 0 with a not unusual rate of convergence, the error in the function values stayed constant. This difficulty did not arise in the new version of `bvpsuite`.

References

[1] B. Swartz De Boor, C. *Collocation at Gaussian points*. 1973.

[2] G. Pulverer Ch. Simon E. Weinmüller G. Kitzhofer, O. Koch. Bvpsuite – a new matlab solver for singular/regular boundary value problems in odes. 2009.

[3] M.L. Morgado Pedro M. Lima. Efficient computational methods for singular free boundary problems using smoothing variable substitutions,. 2012.

[4] M.L. Morgado Pedro M. Lima. Computational approaches to singular free boundary problems in ordinary differential equations. 2014.

[5] Robert D. Russel Uri M. Ascher, Robert M. M. Mattheij. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. 1988.

[6] Dirk Praetorius Winfried Auzinger. *Numerische Mathematik*. 2011.