ASC Report No. 31/2019

Implementation of a pathfollowing strategy with an automatic step-length control: New MATLAB package bvpsuite2.0

W. Auzinger, M. Fallahpour, O. Koch, E.B. Weinmüller

Institute for Analysis and Scientific Computing
Vienna University of Technology — TU Wien
www.asc.tuwien.ac.at ISBN 978-3-902627-00-1

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstraße 8–10
1040 Wien, Austria

E-Mail: `admin@asc.tuwien.ac.at`
WWW: `http://www.asc.tuwien.ac.at`
FAX: +43-1-58801-10196

ISBN 978-3-902627-00-1

Abstract

The main objective of this work was the implementation of a pathfollowing module to complete the MATLAB package `bvpsuite2.0`. In preparation for the implementation, the tangent continuation method was studied, as presented in [7] and [4]. This method was adapted for the already existing code `bvpsuite2.0`. Also the package was adapted where needed for the new function to fit in. Moreover, some features were implemented in the function, that would allow to tackle a broad range of problems. The implementation is tested on various examples.
A secondary objective was the preparation of a manual, which would help first time users and more experienced users of `bvpsuite2.0`, to use the package as intended.
Finally, reports on simulations with `bvpsuite2.0` that were carried out in collaboration with various researchers are given.

Zusammenfassung

In dieser Arbeit war die primäre Zielsetzung ein pathfollowing Modul zu implementieren, um das MATLAB Software Paket `bvpsuite2.0` abzurunden. Aufbauend auf der vorhandenen Theorie der "tangent continuation method" aus [7] und [4], wurde diese angepasst, um der bereits bestehenden Code-Basis von `bvpsuite2.0` zu entsprechen. Der Code wurde auch angepasst, sodass die neue Funktion hineinpasst. Einige zusätzliche Funktionalitäten wurden hinzugefügt, die es ermöglichen sollen, ein breites Spektrum von Problemen anzugehen. Die Implementierung wurde an mehreren Beispielen getestet.
Ein weiteres Ziel der Arbeit war die Erstellung eines Manuals, das Benutzern ermöglichen würde, den Code so zu benutzen wie er konzipiert wurde.
Zuletzt werden die Ergebnisse von Simulationen gezeigt, die im Zuge der Erstellung dieser Arbeit in Zusammenarbeit mit internationalen Kooperationspartnern durchgeführt wurden.

Contents

Chapter 1

Pathfollowing module

In the beginning of this chapter, the existing theoretical framework for pathfollowing along a variable parameter in an algebraic equation with automatic step-length control is presented. Afterwards, the details of the implementation of the pathfollowing module in `bvpsuite2.0` are discussed. Finally, at the end of the chapter, the performance of the implementation is demonstrated on various test examples.
A pathfollowing module was already present in `bvpsuite1.1`, in the form published in [4]. This version was improved by introducing an automatic step-length strategy as described in [6] and [7].

1.1 Theoretical framework

In this section, the general problem setting is introduced. Then, the tangent continuation method, which is used in this work to find approximate homotopy paths numerically, is presented. Some theorems on convergence results of this algorithm are discussed. These results then lead to a theoretically justified step-length control procedure.

1.1.1 General problem setting and requirements

The general problem type considered in this section is

$$\boldsymbol{F}(\boldsymbol{x}, \lambda^*) = \boldsymbol{0}, \qquad \text{with } \; \boldsymbol{x} \in D \subset \mathbb{R}^\eta, \; \lambda^* \in \mathbb{R}, \tag{1.1}$$

where $\eta \in \mathbb{N}$ and $\boldsymbol{F} : \mathbb{R}^{\eta+1} \to \mathbb{R}^\eta$ is a differentiable function. The aim is to follow the evolution of the solution vector $\boldsymbol{x}$ of the implicit equation in (1.1), when the value of λ^* is varied. When following the solutions $\boldsymbol{x}$ of equation (1.1) along a direction, the resulting path is also called the homotopy path. The existence of the homotopy path is given by the implicit function theorem.
Define $\boldsymbol{a} := (\boldsymbol{x}, \lambda^*) \in \mathbb{R}^{\eta+1}$. Then, we require to have

- an isolated homotopy path $\boldsymbol{\mathcal{A}}$ of (1.1), i.e. for all $\boldsymbol{a} \in \boldsymbol{\mathcal{A}}$ we have that $\boldsymbol{F}(\boldsymbol{a}) = \boldsymbol{0}$, where the Jacobian $\boldsymbol{F}'(\boldsymbol{a}) := [\boldsymbol{F}_{\boldsymbol{x}}(\boldsymbol{x}, \lambda^*), \boldsymbol{F}_{\lambda^*}(\boldsymbol{x}, \lambda^*)] \in \mathbb{R}^{\eta \times (\eta+1)}$ has full rank η, and
- an initial solution point $\boldsymbol{a}_0$ on the isolated solution path $\boldsymbol{\mathcal{A}}$.

With these requirements at hand, the tangent continuation method is introduced.

1.1.2 Tangent continuation method

Continuation methods in general consist in moving along the homotopy path, starting from a given solution $\boldsymbol{a}_0 = (\boldsymbol{x}_0, \lambda_0^*)$ and then finding another solution $\boldsymbol{a}_1 = (\boldsymbol{x}_1, \lambda_1^*)$ on the path. The movement along the path is performed in two steps:

1. in the prediction step, a predictor point $\hat{\boldsymbol{a}}_0 \in \mathbb{R}^{\eta+1}$, which is expected to be sufficiently close to the homotopy path, is found and
2. in the correction step, starting from $\hat{\boldsymbol{a}}_0$, the corrected point $\boldsymbol{a}_1$ is iteratively approximated.

In the classical continuation method, the prediction step is performed by changing slightly the value of λ_0^*, but keeping $\boldsymbol{x}$ fixed. The tangent continuation method on the other hand follows the tangent to the homotopy path in the point $\boldsymbol{a}_0$. In this way, it is hoped that the predictor point $\hat{\boldsymbol{a}}_0$ lies closer to the homotopy path than with the classical continuation method. Two steps of the tangent continuation method are displayed schematically in Figure 1.1.

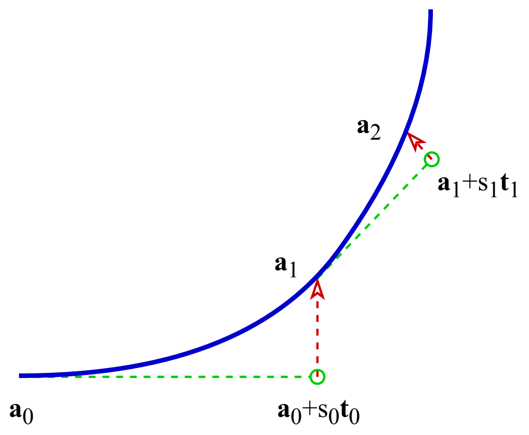

$\mathbf{a}_2$
$\mathbf{a}_1+\mathrm{s}_1\mathbf{t}_1$
$\mathbf{a}_1$
$\mathbf{a}_0$
$\mathbf{a}_0+\mathrm{s}_0\mathbf{t}_0$

Figure 1.1: Schematic tangent continuation method.

In order to compute the tangent $\boldsymbol{t}_0 \in \mathbb{R}^{\eta+1}$ to the homotopy path in the point $\boldsymbol{a}_0$, we augment the function $\boldsymbol{F}$ to $\boldsymbol{F}_1$ by the row $\lambda^* - \lambda_0^*$. This line fixes the value of the variable λ^*, when trying to solve the equation $\boldsymbol{F}_1(\boldsymbol{a}) = \boldsymbol{0}$, to $\lambda_0^* \in \mathbb{R}$ fixed.
The derivative of $\lambda^* - \lambda_0^*$ with respect to $\boldsymbol{x}$ is $\boldsymbol{0}$ and with respect to λ^* it is 1. Therefore the Jacobian of $\boldsymbol{F}_1$ is then equal to the Jacobian $\boldsymbol{F}'$ augmented with the row $(0, ..., 0, 1)$ and it is an $(\eta + 1) \times (\eta + 1)$ square matrix with full rank. Although the kernel of this matrix is trivial, it can be used to compute the tangent $\boldsymbol{t}_0$ of a solution along the homotopy path. The formula used for this task is

$$\boldsymbol{F}_1'(\boldsymbol{a}_0)\boldsymbol{t}_0 = (\boldsymbol{0}, \delta)^T \,, \tag{1.2}$$

where $\delta \in \mathbb{R}$. Due to the shape of the last row of $\boldsymbol{F}_1$, the last entry of $\boldsymbol{t}_0$ is δ. Thus, there exists a $\hat{\boldsymbol{t}}$ such that $\boldsymbol{t}_0 = (\hat{\boldsymbol{t}}, \delta)^T$. The above formula (1.2) is then equivalent to

$$\begin{aligned}
&\begin{pmatrix} \boldsymbol{F}_{\boldsymbol{x}}(\boldsymbol{a}_0) & \boldsymbol{F}_{\lambda^*}(\boldsymbol{a}_0) \\ \boldsymbol{0} & 1 \end{pmatrix} \begin{pmatrix} \hat{\boldsymbol{t}} \\ \delta \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} \\ \delta \end{pmatrix} \\
\Leftrightarrow \quad &\begin{pmatrix} \boldsymbol{F}_{\boldsymbol{x}}(\boldsymbol{a}_0) & \boldsymbol{F}_{\lambda^*}(\boldsymbol{a}_0) \\ \boldsymbol{0} & 1 \end{pmatrix} \begin{pmatrix} \hat{\boldsymbol{t}} \\ 0 \end{pmatrix} + \begin{pmatrix} \boldsymbol{F}_{\boldsymbol{x}}(\boldsymbol{a}_0) & \boldsymbol{F}_{\lambda^*}(\boldsymbol{a}_0) \\ \boldsymbol{0} & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{0} \\ \delta \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} \\ \delta \end{pmatrix} \\
\Leftrightarrow \quad &\begin{pmatrix} \boldsymbol{F}_{\boldsymbol{x}}(\boldsymbol{a}_0) & \boldsymbol{F}_{\lambda^*}(\boldsymbol{a}_0) \\ \boldsymbol{0} & 1 \end{pmatrix} \begin{pmatrix} \hat{\boldsymbol{t}} \\ 0 \end{pmatrix} + \delta \begin{pmatrix} \boldsymbol{F}_{\lambda^*}(\boldsymbol{a}_0) \\ 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} \\ \delta \end{pmatrix} \\
\Leftrightarrow \quad &\boldsymbol{F}_{\boldsymbol{x}}(\boldsymbol{a}_0)\, \hat{\boldsymbol{t}} + \boldsymbol{F}_{\lambda^*}(\boldsymbol{a}_0)\, \delta = \boldsymbol{0}.
\end{aligned}$$

This is then an overdetermined system of equations, the same system as the one used in [4] to compute the tangent. When λ is set to 1, then this system has a unique solution $\hat{t}$, provided $\boldsymbol{F}_{\boldsymbol{x}}(\boldsymbol{a}_0)$ is a regular matrix.
After computing the tangent $\boldsymbol{t}_0$ by the formula (1.2), the tangent is normalized, i.e. $\|\boldsymbol{t}_0\|_2 = 1$ in the Euclidean norm. Due to these requirements, $\boldsymbol{t}_0$ is uniquely determined up to orientation and the choice of setting δ to 1 does not affect the outcome.
Next, a step-length $s \in \mathbb{R}^+$ is chosen and the predictor point is computed as $\hat{\boldsymbol{a}}_0 := \boldsymbol{a}_0 + s \cdot \boldsymbol{t}_0$. This then completes the prediction step of the tangent continuation method. Regarding the choice of the step-length, the details are mentioned in Section 1.1.4.

The correction step consists of an iterative scheme, for instance the Newton iteration or a variant of it. The correction step starts at the predictor point $\hat{\boldsymbol{a}}_0$, when chosen well, the scheme converges towards the corrected point $\boldsymbol{a}_1$, a point on the homotopy path.
Similarly to the computation of the tangent in the prediction step, in the correction step, the function $\boldsymbol{F}$ is augmented by a row to $\boldsymbol{F}_2$, where the last row of $\boldsymbol{F}_2$ ensures that every solution $\hat{\boldsymbol{a}}$ of $\boldsymbol{F}(\hat{\boldsymbol{a}}) = \boldsymbol{0}$ is orthogonal to the tangent and passing through $\hat{\boldsymbol{a}}_0$.
Thus $\boldsymbol{F}_2$ has the form

$$\boldsymbol{F}_2(\hat{\boldsymbol{a}}) := \begin{pmatrix} \boldsymbol{F}(\boldsymbol{a}) \\ (\hat{\boldsymbol{a}} - \boldsymbol{a}_0) \cdot \boldsymbol{t}_0 - s \end{pmatrix}.$$

The Jacobian of this matrix is

$$\boldsymbol{F}_2'(\hat{\boldsymbol{a}}) := \begin{pmatrix} \boldsymbol{F}_{\boldsymbol{x}}(\hat{\boldsymbol{a}}) \quad \boldsymbol{F}_{\lambda^*}(\hat{\boldsymbol{a}}) \\ \boldsymbol{t}_0 \end{pmatrix}.$$

The form of the function $\boldsymbol{F}_2$ marks the main difference between the pathfollowing algorithm presented by Deuflhard et al. in [6] and [7] and the algorithm implemented in `bvpsuite1.1` and published in [4]. While Deuflhard works with non-quadratic matrices and a Gauss-Newton method to follow the homotopy path beyond turning points, Kitzhofer et al. use the approach of square matrices and the Fast Frozen Newton method.

In this work also, the method of interest is the Fast Frozen Newton method, as it is implemented as a solver routine for nonlinear problems in `bvpsuite2.0` ([2]). This method is a refined version of the simplified Newton iteration, which in turn is a simplification of the classical Newton method. Therefore all three methods are briefly introduced here.
The Newton iteration starts at an initial point $\hat{\boldsymbol{a}}_0 \in \mathbb{R}^{\eta+1}$ and produces a sequence $\{\hat{\boldsymbol{a}}_k\}_{k\in\mathbb{N}_0}$, such that $\hat{\boldsymbol{a}}_k \to \boldsymbol{a}_1$, which is a root of the function $\boldsymbol{F}_2$. In this procedure, for each $k \in \mathbb{N}_0$, the Newton increment $\Delta\hat{\boldsymbol{a}}_k$ is computed as the solution of the equation

$$\boldsymbol{F}_2'(\hat{\boldsymbol{a}}_k)\Delta\hat{\boldsymbol{a}}_k = -\boldsymbol{F}_2(\hat{\boldsymbol{a}}_k).$$

and the next approximation is computed as $\hat{\boldsymbol{a}}_{k+1} = \hat{\boldsymbol{a}}_k + \Delta\hat{\boldsymbol{a}}_k$. A convergence result for such a method can be found in Theorem 2.2 in [7, pp. 49–50].
In the simplified Newton method, as presented in [7], the increment is once computed as above and in the subsequent steps the simplified increment $\overline{\Delta\hat{\boldsymbol{a}}_{k+1}}$ is computed from the equation

$$\boldsymbol{F}_2'(\hat{\boldsymbol{a}}_0)\overline{\Delta\hat{\boldsymbol{a}}_k} = -\boldsymbol{F}_2(\hat{\boldsymbol{a}}_{k+1}).$$

This means that the simplified increment is computed by evaluating the inverse of the Jacobian at the first iterate $\hat{\boldsymbol{a}}_0$ instead of at the updated approximation $\hat{\boldsymbol{a}}_{k+1}$. This measure saves computation time, because in this case the computation of the increment reduces to a matrix-vector multiplication of $\boldsymbol{F}_2'(\boldsymbol{a}_0)^{-1}$ and $\boldsymbol{F}_2(\boldsymbol{a}_k)$[1]. The convergence properties of the simplified Newton method are discussed in Section 1.1.3.
The fast frozen Newton method can be considered a mixture of these two methods, the Jacobian is updated, but not in every step. In fact, it is only updated, when

$$\|\overline{\Delta\hat{\boldsymbol{a}}_{k+1}}\| > \xi\|\overline{\Delta\hat{\boldsymbol{a}}_k}\|, \tag{1.3}$$

where $\xi < 1$ – by default $\xi = 0.5$ in `bvpsuite2.0`. Hence, the fast frozen Newton method only updates the Jacobian whenever the convergence is slow. Otherwise it keeps the Jacobian “frozen” for a number of steps until the inequality above is satisfied.
As mentioned above, the choice of the step-length is crucial for the convergence of the Newton methods, due to their restricted domain of convergence. This is the reason why the step-length adaptation procedure detailed in Section 1.1.4 is needed.

The tangent continuation method as presented above is theoretically able to follow paths beyond turning points, but not bifurcation points. As presented, the method does not account for cases in which the Jacobian $\boldsymbol{F}'$ is non-singular, which is the case whenever passing a bifurcation point. This is a weakness of this method presented here, but in some examples containing bifurcation points this does not pose any problem, as can be seen in Section 1.3. This may be due to the implementation of the backslash operator in MATLAB, which through its least squares approach sometimes finds a satisfactory tangent regardless of the rank of the Jacobian. Possibly in the examples the bifurcation points

[1]Note, that in `bvpsuite2.0` the inverse of the Jacobian is never explicitly computed, instead the *LU*-decomposition of $\boldsymbol{F}_2'(\boldsymbol{a}_0)$ is saved and reused in the non-linear solver.

are overlooked, because the step-length was long enough to pass by such a singularity. The convergence properties of the simplified Newton method as introduced above will be the next subject of discussion. The results are needed to formulate the mentioned adaptive step-length control strategy.

1.1.3 About the simplified Newton method

The theorems in this section were taken from [6, pp. 52–54 and 253–257]. These theorems are slightly adapted to fit the case of square matrix problems, as these are the tools used in this work.
We concentrate in the convergence analysis in this chapter on the simplified Newton method as introduced above, for a function $\boldsymbol{G} : \mathbb{R}^n \to \mathbb{R}^n$ with $n \in \mathbb{N}$,

$$\boldsymbol{G}'(\boldsymbol{a}_0)\overline{\Delta \boldsymbol{a}_k} = -\boldsymbol{G}(\boldsymbol{a}_k), \quad \boldsymbol{a}_{k+1} = \boldsymbol{a}_k + \overline{\Delta \boldsymbol{a}_k}, \quad \text{for } \{\boldsymbol{a}_k\}_{k\in\mathbb{N}_0} \subset \mathbb{R}^n. \tag{1.4}$$

The simplified Newton method will approach iteratively a solution $\boldsymbol{a}_*$ of the implicit equation $\boldsymbol{G}(\boldsymbol{a}) = 0$ if the starting point of the iteration is chosen sufficiently close to $\boldsymbol{a}_*$.

Theorem 1. *Let $\boldsymbol{G} : D \subset \mathbb{R}^n \to \mathbb{R}^n$ denote some C^1-mapping with D open and convex. Let $\boldsymbol{a}_0 \in D$ denote a given starting point so that $\boldsymbol{G}'(\boldsymbol{a}_0)$ is invertible. Assume the Lipschitz condition*

$$\|\boldsymbol{G}'(\boldsymbol{a}_0)^{-1}(\boldsymbol{G}'(\boldsymbol{a}) - \boldsymbol{G}'(\boldsymbol{a}_0))\| \leq \omega_0\|\boldsymbol{a} - \boldsymbol{a}_0\| \tag{1.5}$$

to hold, for all $\boldsymbol{a} \in D$. Let

$$h_0 := \omega_0\|\overline{\Delta \boldsymbol{a}_0}\| \leq \frac{1}{2}$$

and define

$$t^* := t - \sqrt{1 - 2h_0} \quad \text{and} \quad \rho := \frac{t^*}{\omega_0}.$$

Moreover, assume that the closed ball $\overline{B}(\boldsymbol{a}_0, \rho)$ with center $\boldsymbol{a}_0$ and radius ρ is in D. Then the iterates $\boldsymbol{a}_k$ of the simplified Newton method (1.4) *remain in $\overline{B}(\boldsymbol{a}_0, \rho)$ and converge to some $\boldsymbol{a}_*$ with $\boldsymbol{G}(\boldsymbol{a}_*) = \boldsymbol{0}$.*
The sequence $\{\boldsymbol{a}_k\}_{k\in\mathbb{N}}$ converges linearly, i.e.

$$\lim_{k\to\infty} \frac{\|\boldsymbol{a}_{k+1} - \boldsymbol{a}_*\|}{\|\boldsymbol{a}_k - \boldsymbol{a}_*\|} = c < 1.$$

Proof. With the Lipschitz condition (1.5) we have

$$\begin{aligned}
\|\boldsymbol{a}_{k+1} - \boldsymbol{a}_k\| &= \|\overline{\Delta \boldsymbol{a}_k}\| = \|\boldsymbol{G}'(\boldsymbol{a}_0)^{-1}\boldsymbol{G}(\boldsymbol{a}_k)\| \\
&= \|\boldsymbol{G}'(\boldsymbol{a}_0)^{-1}\left[\boldsymbol{G}(\boldsymbol{a}_k) + \boldsymbol{G}(\boldsymbol{a}_{k-1}) - \boldsymbol{G}(\boldsymbol{a}_{k-1})\right]\| \\
&= \|\boldsymbol{G}'(\boldsymbol{a}_0)^{-1}\left[\boldsymbol{G}(\boldsymbol{a}_{k-1} + \overline{\Delta \boldsymbol{a}_{k-1}}) - \boldsymbol{G}'(\boldsymbol{a}_{k-1})\overline{\Delta \boldsymbol{a}_{k-1}} - \boldsymbol{G}(\boldsymbol{a}_{k-1}) + \boldsymbol{0}\right]\|
\end{aligned}$$

$$
\begin{aligned}
&= \|\boldsymbol{G}'(\boldsymbol{a}_0)^{-1} \int_0^1 \left[\boldsymbol{G}'(\boldsymbol{a}_{k-1} + \gamma\overline{\Delta\boldsymbol{a}_{k-1}}) - \boldsymbol{G}'(\boldsymbol{a}_{k-1})\right] \overline{\Delta\boldsymbol{a}_{k-1}}\,\mathrm{d}\gamma\,\| \\
&\leq \| \int_0^1 \omega_0\gamma(\overline{\Delta\boldsymbol{a}_{k-1}})^2\,\mathrm{d}\gamma\,\| = \frac{1}{2}\omega_0\|\overline{\Delta\boldsymbol{a}_{k-1}}\|^2 = \frac{1}{2}\omega_0\|\boldsymbol{a}_k - \boldsymbol{a}_{k-1}\|^2 \\
&\leq \frac{1}{2}\omega_0\|\boldsymbol{a}_k - \boldsymbol{a}_{k-1}\| \left(\|\boldsymbol{a}_{k-1} - \boldsymbol{a}_0\| + \|\boldsymbol{a}_k - \boldsymbol{a}_0\|\right). \qquad (1.6)
\end{aligned}
$$

The upper bound sequences $\{h_k\}_{k\in\mathbb{N}_0}$ and $\{t_k\}_{k\in\mathbb{N}_0}$ are introduced with the respective properties

$$\omega_0\|\boldsymbol{a}_{k+1} - \boldsymbol{a}_k\| \leq h_k, \quad \omega_0\|\boldsymbol{a}_k - \boldsymbol{a}_0\| \leq t_k$$

and with initial values $t_0 = 0$ and $h_0 = \omega_0\|\overline{\Delta\boldsymbol{a}_0}\| \leq \frac{1}{2}$. Then, we choose

$$t_{k+1} := t_k + h_k, \quad h_k := \frac{1}{2}h_{k-1}(t_k + t_{k-1}),$$

because with the inequality (1.6) above

$$\omega_0\|\boldsymbol{a}_{k+1} - \boldsymbol{a}_k\| \leq \frac{1}{2}h_{k-1}(t_k + t_{k-1})$$

and with the triangle inequality

$$\omega_0\|\boldsymbol{a}_{k+1} - \boldsymbol{a}_0\| \leq \omega_0\|\boldsymbol{a}_k - \boldsymbol{a}_0\| + \omega_0\|\boldsymbol{a}_{k+1} - \boldsymbol{a}_k\| \leq t_k + h_k.$$

Then the equation

$$t_{k+1} - t_k = h_k = \frac{1}{2}h_{k-1}(t_k + t_{k-1}) = \frac{1}{2}(t_k^2 + t_{k-1}^2)$$

follows. This leads to

$$t_{k+1} - \frac{1}{2}t_k^2 = t_k - \frac{1}{2}t_{k-1}^2 \quad \Rightarrow \quad t_{k+1} - \frac{1}{2}t_k^2 = ... = t_1 - \frac{1}{2}t_0^2 = h_0.$$

This can be rewritten as the simplified scalar Newton iteration

$$t_{k+1} - t_k = -\frac{g(t_k)}{g'(t_0)} = g(t_k) \quad \text{for the equation} \quad g(t) := h_0 - t + \frac{1}{2}t^2 = 0.$$

$g(t) = 0$ has the solutions $t = 1 \pm \sqrt{1 - 2h_0}$. Here the condition $h_0 < \frac{1}{2}$ is used.
Consider the sequence $t_{k+1} = t_k + g(t_k) = h_0 + \frac{1}{2}t_k^2$ for $k \in \mathbb{N}_0$ and $t_0 = 0$. By induction, $t_{k+1} > t_k$ and because the derivative $g'(t) = t - 1$ is negative for $t < 1$, $g(t_{k+1}) < g(t_k)$ also holds, which is equivalent to $h_{k+1} < h_k$. Also, $t_{k+1}^2 > t_k^2$ and thus $g(t_{k+1}) = -\frac{1}{2}t_k^2 + \frac{1}{2}(h_0 + \frac{1}{2}t_k^2)^2 > 0$. This leads to the property

$$t_k \leq t^* = 1 - \sqrt{1 - 2h_0}.$$

Therefore, the iteration t_k starting at $t_0 = 0$ converges towards the root $1 - \sqrt{1 - 2h_0}$. This implies $\boldsymbol{a}_k \in \overline{S}(\boldsymbol{a}_0, \rho) \subset D$. The property $h_{k+1} < h_k$ implies $\{\boldsymbol{a}_k\}_{k \in \mathbb{N}_0}$ to be a Cauchy sequence. Thus there exists $\boldsymbol{a}_* \in \overline{S}(\boldsymbol{a}_0, \rho)$ with $\boldsymbol{a}_k \to \boldsymbol{a}_*$ for $k \to \infty$. Since $\|\overline{\Delta \boldsymbol{a}_k}\| = \|\boldsymbol{a}_{k+1} - \boldsymbol{a}_k\| \to 0$ and $\boldsymbol{G}'(\boldsymbol{a}_0)\overline{\Delta \boldsymbol{a}_k} = -\boldsymbol{G}(\boldsymbol{a}_k)$, $\boldsymbol{G}(\boldsymbol{a}_*) = \boldsymbol{0}$ holds.
For the convergence rate, note that with the use of the triangle inequality

$$t_k = \sum_{i=0}^{k} h_i \quad \Rightarrow \quad \omega_0 \|\boldsymbol{a}_k - \boldsymbol{a}_*\| \leq \sum_{i=k}^{\infty} h_i = t^* - t_k.$$

Furthermore, with the use of (1.5) and $\boldsymbol{G}(\boldsymbol{a}_*) = \boldsymbol{0}$, it holds

$$\begin{aligned}
\|\boldsymbol{a}_{k+1} - \boldsymbol{a}_*\| &= \|\boldsymbol{a}_k + \overline{\Delta \boldsymbol{a}_k} - \boldsymbol{a}_*\| = \|\boldsymbol{a}_k - \boldsymbol{a}_* - \boldsymbol{G}'(\boldsymbol{a}_k)^{-1}(\boldsymbol{G}(\boldsymbol{a}_k) - \boldsymbol{G}(\boldsymbol{a}_*))\| \\
&= \|\boldsymbol{G}'(\boldsymbol{a}_k)^{-1}\left[(\boldsymbol{G}(\boldsymbol{a}_*) - \boldsymbol{G}(\boldsymbol{a}_k)) + \boldsymbol{G}'(\boldsymbol{a}_k)(\boldsymbol{a}_k - \boldsymbol{a}_*)\right]\| \\
&= \|\boldsymbol{G}'(\boldsymbol{a}_k)^{-1} \int_0^1 (\boldsymbol{G}'(\boldsymbol{a}_k + \gamma(\boldsymbol{a}_k - \boldsymbol{a}_*)) - \boldsymbol{G}'(\boldsymbol{a}_*))(\boldsymbol{a}_k - \boldsymbol{a}_*)\,\mathrm{d}\gamma\| \\
&\leq \omega_0 \|\boldsymbol{a}_k - \boldsymbol{a}_*\|^2 \leq (t^* - t_k)\|\boldsymbol{a}_k - \boldsymbol{a}_*\|.
\end{aligned}$$

Since $t^* - t_k < 1$, the sequence $\{\boldsymbol{a}_k\}_{k \in \mathbb{N}}$ converges linearly. □

Having established the convergence of the simplified Newton method to an element which satisfies the implicit equation given by $\boldsymbol{G}$, this result will be used to prove convergence starting from a point given by the tangent continuation method.

For the tangent continuation method, a starting point $\boldsymbol{a}_0 \in \mathbb{R}^n$ on the homotopy path, a step-length $s \in \mathbb{R}$ and the normalized tangent vector $\boldsymbol{t}_0 \in \mathbb{R}^n$ to the homotopy path at $\boldsymbol{a}_0$ are needed. The correction method starts at the prediction point

$$\hat{\boldsymbol{a}}_0 := \boldsymbol{a}_0 + s \cdot \boldsymbol{t}_0.$$

The theorem below implies a restriction on s, for which the simplified Newton method converges to the homotopy path.
In order to relate these theoretical results to the implementation in `bvpsuite2.0`, some details are adjusted. The function $\boldsymbol{G} : D \subset \mathbb{R}^n \to \mathbb{R}^n$ denotes some C^1-mapping with D open, convex, and sufficiently large. Note that with this function $\boldsymbol{G}$, any $0 \neq \boldsymbol{a} \in \mathbb{R}^n$ is considered to be on the homotopy path if and only if $\boldsymbol{G}(\boldsymbol{a}) = (0, ..., 0, r)$ for $r \in \mathbb{R}$. The n-th row of $\boldsymbol{G}(\boldsymbol{a})$ is $(\boldsymbol{a} - \boldsymbol{a}_0) \cdot \boldsymbol{t}_0 - s$, which implies the n-th row of the Jacobian $\boldsymbol{G}'(\boldsymbol{a})$ to be simply $\boldsymbol{t}_0$. Thus

$$\boldsymbol{G}(\boldsymbol{a}_0) = (0, ..., 0, -s)^T \quad \text{and} \quad \boldsymbol{G}'(\boldsymbol{a}_0)\boldsymbol{t}_0 = (0, ..., 0, 1)^T. \tag{1.7}$$

Finally, let

$$H := \{\boldsymbol{a} \in \mathbb{R}^n \mid (\boldsymbol{a} - \boldsymbol{a}_0) \cdot \boldsymbol{t}_0 - s = 0\} = \{\boldsymbol{a} \in \mathbb{R}^n \mid \boldsymbol{a} = \boldsymbol{a}_0 + s \cdot \boldsymbol{t}_0 + r \text{ with } r \perp \boldsymbol{t}_0\}.$$

Then the following theorem can be stated.

Theorem 2. *Let $\boldsymbol{a}_0$, s, $\boldsymbol{t}_0$, $\hat{\boldsymbol{a}}_0$, $\boldsymbol{G}$, D and H be as above. Then, under the assumption of the Lipschitz conditions*

$$\left\|\boldsymbol{G}'(\boldsymbol{a}_0)^{-1}\left(\boldsymbol{G}'(\boldsymbol{a})-\boldsymbol{G}'(\boldsymbol{a}_0)\right)\right\| \leq \omega_0\|\boldsymbol{a}-\boldsymbol{a}_0\|, \quad \textit{for } \boldsymbol{a},\boldsymbol{a}_0 \in H \tag{1.8}$$

and

$$\left\|\boldsymbol{G}'(\boldsymbol{a})^{-1}\left(\boldsymbol{G}'(u+\delta_2 s\cdot\boldsymbol{t}(u))-\boldsymbol{G}'(u)\right)\boldsymbol{t}(u)\right\| \leq \delta_2\omega_t\|\boldsymbol{t}(u)\|_2^2, \tag{1.9}$$

with the normalized tangential vector $\boldsymbol{t}(u)$ to the homotopy path at u and $\boldsymbol{a}$, $u+\delta_2 s\cdot\boldsymbol{t}(u) \in D$, with $0 \leq \delta_2 \leq 1$. Then the simplified Newton iteration converges for all

$$s \leq s_{\max} := \frac{1}{\sqrt{\omega_0\omega_t}}. \tag{1.10}$$

Proof. For the simplified Newton iteration in H we may apply Theorem 1, which here requires the verification of the sufficient condition

$$\|\overline{\Delta\hat{\boldsymbol{a}}_0}\|\omega_0 \leq \alpha_0(s)\omega_0 \leq \frac{1}{2}.$$

Then the derivation of an appropriate $\alpha_0(s)$ may proceed with (1.7) and (1.9) as

$$\begin{aligned}
\|\overline{\Delta\hat{\boldsymbol{a}}_0}\| &= \|\boldsymbol{G}'(\hat{\boldsymbol{a}}_0)^{-1}\boldsymbol{G}(\hat{\boldsymbol{a}}_0)\| = \|\boldsymbol{G}'(\hat{\boldsymbol{a}}_0)^{-1}\left[\boldsymbol{G}(\hat{\boldsymbol{a}}_0)-\boldsymbol{G}(\boldsymbol{a}_0)-(0,...,0,s)^T+0\right]\| \\
&= \|\boldsymbol{G}'(\hat{\boldsymbol{a}}_0)^{-1}\int_{\delta=0}^{s}\left[\boldsymbol{G}'(\boldsymbol{a}_0+\delta\boldsymbol{t}_0)\boldsymbol{t}_0-(0,...,0,2)^T\right]\mathrm{d}\delta\,\| \\
&\leq \int_{\delta=0}^{s}\|\boldsymbol{G}'(\hat{\boldsymbol{a}}_0)^{-1}\left[\boldsymbol{G}'(\boldsymbol{a}_0+\delta\boldsymbol{t}_0)\boldsymbol{t}_0-(0,...,0,1)^T\right]\|\,\mathrm{d}\delta \\
&= \int_{\delta=0}^{s}\|\boldsymbol{G}'(\hat{\boldsymbol{a}}_0)^{-1}\left[\boldsymbol{G}'(\boldsymbol{a}_0+\delta\boldsymbol{t}_0)-\boldsymbol{G}'(\boldsymbol{a}_0)\right]\boldsymbol{t}_0\|\,\mathrm{d}\delta \leq \frac{1}{2}\omega_t s^2.
\end{aligned}$$

Hence

$$\alpha_0(s) := \frac{1}{2}\omega_t s^2, \tag{1.11}$$

which inserted above directly leads to the maximum feasible stepsize $s_{\max}$. □

Having an upper bound for the step-length for the convergence of the simplified Newton method starting from the prediction point $\hat{\boldsymbol{a}}$, the focus turns now to the prediction of a new step-length.

Let $\boldsymbol{t}(\boldsymbol{a})$ denote the tangent to the path in each point $\boldsymbol{a} \in \mathcal{Y}$ of the homotopy path and again $\boldsymbol{a}_0$ be a point on the homotopy path with $\boldsymbol{t}_0 := \boldsymbol{t}(\boldsymbol{a}_0)$. Let $\hat{\boldsymbol{a}} : \mathbb{R}_0^+ \to \mathbb{R}^n, s \mapsto \boldsymbol{a}_0 + s\cdot\boldsymbol{t}_0$ be a function. Then the point $\boldsymbol{a}(s)$ is defined as the intersection of the homotopy path and the hyperplane orthogonal to $\boldsymbol{t}_0$, passing through $\hat{\boldsymbol{a}}(s)$. Finally, define

$$c_s := \boldsymbol{t}_0^T\boldsymbol{t}(\boldsymbol{a}(s)). \tag{1.12}$$

Consider

$$\begin{aligned} & z(s) := \boldsymbol{a}(s) - \hat{\boldsymbol{a}}(s) \\ \Rightarrow \quad & \dot{z}(s) = \dot{\boldsymbol{a}}(s) - \boldsymbol{t}_0, \\ \Rightarrow \quad & \ddot{z}(s) = \ddot{\boldsymbol{a}}(s). \end{aligned}$$

Similarly to above, for a fixed $\overline{s} \in \mathbb{R}^n$, $\boldsymbol{G} : \mathbb{R}^n \to \mathbb{R}^n$ is a C^2-function in $\boldsymbol{a}$ with last row entry $(\boldsymbol{a} - \boldsymbol{a}(0)) \cdot \boldsymbol{t}_0 - \overline{s}$. For $s \in \mathbb{R}_0^+$, $\boldsymbol{a}(s) = \boldsymbol{a}_0 + s \cdot \boldsymbol{t}_0 + r$ with $r \perp \boldsymbol{t}_0$. Thus

$$\boldsymbol{G}(\overline{\boldsymbol{a}}(s)) \equiv (0, ..., 0, s - \overline{s})^T,$$

$$\Rightarrow \quad \boldsymbol{G}'(\boldsymbol{a}(s))\dot{\boldsymbol{a}}(s) \equiv (0, ..., 0, 1)^T, \tag{1.13}$$

$$\Rightarrow \quad \boldsymbol{G}''(\boldsymbol{a}(s)) \left[\dot{\boldsymbol{a}}(s)\right]^2 + \boldsymbol{G}'(\boldsymbol{a}(s))\ddot{\boldsymbol{a}}(s) \equiv \boldsymbol{0}. \tag{1.14}$$

From (1.13) it follows

$$\dot{\boldsymbol{a}}(s) = \frac{1}{c_s}\boldsymbol{t}(\boldsymbol{a}(s)),$$

since $\boldsymbol{G}'(\boldsymbol{a}(s))\boldsymbol{t}(\boldsymbol{a}(s)) = (0, ...0, c_s)^T$. Thus $\dot{\boldsymbol{a}}(0) = \boldsymbol{t}_0$. Consider now

$$\|z(s)\| = \|\boldsymbol{a}(s) - \hat{\boldsymbol{a}}(s)\| = \|\boldsymbol{a}(s) - \boldsymbol{a}(0) - s \cdot \dot{\boldsymbol{a}}(0)\| = \left\| \int_0^1 \dot{\boldsymbol{a}}(\gamma s) s \, \mathrm{d}\gamma - s \cdot \dot{\boldsymbol{a}}(0) \right\| \tag{1.15}$$

$$= s\| \underbrace{\max_{\gamma \in [0,s]} \dot{\boldsymbol{a}}(\gamma)}_{=: \dot{\boldsymbol{a}}(\gamma^*)} - \dot{\boldsymbol{a}}(0)\| = s \left\| \int_0^{\gamma^*} \ddot{\boldsymbol{a}}(\gamma s) s \, \mathrm{d}\gamma \right\| = s^2 \max_{\gamma \in [0,s]} \|\ddot{\boldsymbol{a}}(s)\|. \tag{1.16}$$

The following theorem gives an upper bound for $\|\ddot{\boldsymbol{a}}(s)\|$, $s \in \mathbb{R}_0^+$.

Theorem 3. *Let $\boldsymbol{t}(\cdot)$, $\boldsymbol{t}_0$, $\hat{\boldsymbol{a}}(\cdot)$, $\boldsymbol{a}(\cdot)$, c_s, $\overline{s}$ and $\boldsymbol{G}$ be as above. The inequalities* (1.8) *and* (1.9) *are satisfied as in the preceding theorem. Then, for $c_s \neq 0$, one has*

$$\|\ddot{\boldsymbol{a}}(s)\|_2 \leq \frac{\omega_t}{c_s^2}. \tag{1.17}$$

Proof. From (1.14) it follows

$$\ddot{\boldsymbol{a}}(s) = \frac{1}{c_s^2}\boldsymbol{G}'(\boldsymbol{a}(s))^{-1}\boldsymbol{G}''(\boldsymbol{a}(s)) \left[\boldsymbol{t}(\boldsymbol{a}(s))\right]^2.$$

With the relation

$$\lim_{\delta_2 \to 0} \frac{1}{\delta_2} \left(\boldsymbol{G}'(\boldsymbol{a}(s) + \delta_2 \boldsymbol{t}(\boldsymbol{a}(s))) - \boldsymbol{G}'(\boldsymbol{a}(s))\right) = \boldsymbol{G}''(\boldsymbol{a}(s))\boldsymbol{t}(\boldsymbol{a}(s))$$

and by application of (1.9), it follows

$$\|\ddot{\boldsymbol{a}}(s)\| = \frac{1}{c_s^2} \left\| \boldsymbol{G}'(\boldsymbol{a}(s))^{-1} \lim_{\delta_2 \to 0} \frac{1}{\delta_2} \left(\boldsymbol{G}'(\boldsymbol{a}(s) + \delta_2 \boldsymbol{t}(\boldsymbol{a}(s))) - \boldsymbol{G}'(\boldsymbol{a}(s))\right) \boldsymbol{t}(\boldsymbol{a}(s)) \right\|$$

$$= \frac{1}{c_s^2} \lim_{\delta_2 \to 0} \frac{1}{\delta_2} \|\boldsymbol{G}'(\boldsymbol{a}(s))^{-1} \left(\boldsymbol{G}'(\boldsymbol{a}(s) + \delta_2 \boldsymbol{t}(\boldsymbol{a}(s))) - \boldsymbol{G}'(\boldsymbol{a}(s))\right) \boldsymbol{t}(\boldsymbol{a}(s))\|$$
$$\leq \frac{1}{c_s^2} \lim_{\delta_2 \to 0} \frac{1}{\delta_2} \delta_2 \omega_t \|\boldsymbol{t}(\overline{\boldsymbol{a}}(s))\| \leq \frac{\omega_t}{c_s^2}.$$

This completes the proof. □

With these results in hand, it is now possible to proceed to formalize an adaptive step-length control strategy.

1.1.4 Adaptive step-length control strategy

The work of Deuflhard et al. from [7], [6], and also [5], is the original source of the strategy explained below.

The way the theoretical results from the previous section are used, is to replace the unavailable Lipschitz-constants ω_0 and ω_t by available local estimates $[\omega_0]$ and $[\omega_t]$, which on the basis of the theoretical maximal step-length $s_{\max}$ from Theorem 2 leads to

$$[s_{\max}] := \frac{1}{\sqrt{[\omega_0][\omega_t]}}. \tag{1.18}$$

Since $[\omega_0]$ and $[\omega_t]$ are lower-bounds of ω_0 and ω_t respectively, it follows that $[s_{\max}] \geq s_{\max}$. Therefore in the automatic step-length procedure, a prediction strategy, where a suitable step-length is suggested, and a correction strategy, where this step-length might get adjusted, are needed.

The correction strategy will be applied whenever for a point $\boldsymbol{a}_0$ on the homotopy path with tangent $\boldsymbol{t}_0$ to the path in this point a step-length s_0 is chosen and thus a prediction point $\hat{\boldsymbol{a}}_0 := \boldsymbol{a}_0 + s \cdot \boldsymbol{t}_0$ is found. The correction strategy will reduce the size of the step-length, if specific criteria are not met. Also, the step-length may be elongated if the criteria are too amply satisfied. The aim is to have an efficient process.
The first criterion consists in the first contraction factor θ_0, which gives some knowledge about the convergence of the simplified Newton method in the first step. From (1.6) in the proof of Theorem 1, it follows

$$\|x_{k+1} - x_k\| \leq \frac{1}{2}\|x_k - x_{k-1}\| \left(t_k + t_{k-1}\right) \Rightarrow \frac{\|\overline{\Delta x_{k+1}}\|}{\|\overline{\Delta x_k}\|} \leq \frac{1}{2}\left(t_k + t_{k-1}\right).$$

As in the proof, $t_0 = 0$ and $t_1 = h_0 = \omega_0 \|\overline{\Delta x_0}\| \leq \frac{1}{2}$. Thus, the first condition that needs to be satisfied to achieve convergence of the correction method is

$$\theta_0 := \frac{\|\overline{\Delta x_1}\|}{\|\overline{\Delta x_0}\|} \leq \frac{1}{4} =: \theta_{\max}. \tag{1.19}$$

Therefore, starting the iteration from the predictor point $\hat{\boldsymbol{a}}_0$, from the inequalities above there follows the estimate

$$\omega_0 \geq \frac{2\|\overline{\Delta \hat{\boldsymbol{a}}_{0,1}}\|}{\|\overline{\Delta \hat{\boldsymbol{a}}_0}\|} = \frac{2\Theta_0}{\|\overline{\Delta \hat{\boldsymbol{a}}_0}\|} =: [\omega_0] \tag{1.20}$$

and in the same way from (1.11)

$$\omega_t = \frac{2\alpha_0(s_0)}{s_0^2} \geq \frac{2\|\overline{\Delta \hat{\boldsymbol{a}}_0}\|}{s_0^2} =: [\omega_t]. \tag{1.21}$$

Inserting these two local estimates into (1.18) gives

$$s_0' := \sqrt{\frac{\theta_{\max}}{\theta_0}} s_0. \tag{1.22}$$

This is the step-length suggestion which is applied, whenever the condition $\theta_0 \leq \theta_{\max}$ is not satisfied.
As mentioned above, it is also possible to elongate the step-length, for instance whenever θ_0 is smaller than a predetermined $\theta_{min} < \frac{1}{4}$. The formula (1.22) can then be applied until

$$\theta_{min} \leq \theta_0 \leq \theta_{\max} \tag{1.23}$$

is satisfied. The elongation of the step-length is not activated per default in the implementation, since caution is favoured over efficiency in the pathfollowing module.

Once the iteration converged from $\hat{\boldsymbol{a}}_0$ to a new point $\boldsymbol{a}_1$ on the homotopy path with tangent $\boldsymbol{t}_1$ to the path in this point, then the prediction strategy suggests a new step-length s_1 for the computation of a prediction point $\hat{\boldsymbol{a}}_1 := \boldsymbol{a}_1 + s_1 \cdot \boldsymbol{t}_1$, from where the simplified Newton method will start and is expected to converge. In case the step-length does not fully satisfy the requirements of the correction strategy, then it is adjusted further as is explained above.
In the correction strategy, a step-length s_0 is already known in the beginning of the process. This step-length then gets adapted, in potentially a few steps, to a new step-length s_0'. Here in the correction strategy, the starting guess for the prediction is the step-length s_0'. Combining the inequalities (1.15) and from Theorem 3 (1.17), then

$$\|\hat{\boldsymbol{a}}_0 - \boldsymbol{a}_1\| \leq s_0'^2 \frac{\omega_t}{c_{s_0'}^2}.$$

This leads to the estimate

$$[\omega_t] := \frac{c_{s_0'}^2 \|\hat{\boldsymbol{a}}_0 - \boldsymbol{a}_1\|}{s_0'^2} \leq \omega_t. \tag{1.24}$$

Then, inserting the two estimates $[\omega_0]$ as in (1.20) and $[\omega_t]$ as in (1.24) into the equation

(1.18) results in the step-length prediction formula

$$s_1 := \left(\frac{2}{c^2_{s'_0}} \frac{\theta_{\max}}{\theta_0} \frac{\|\overline{\Delta \hat{\boldsymbol{a}}_0}\|}{\|\hat{\boldsymbol{a}}_0 - \boldsymbol{a}_1\|} \right)^{\frac{1}{2}} s'_0. \tag{1.25}$$

The application of this formula completes the prediction strategy.

This procedure is expected to give good step-lengths for the pathfollowing of a problem. This is backed by the theoretical results presented in the previous sections. In the following sections, the focus will be shifted from the theory to the implementation of this whole algorithm. The tangent continuation method with the adaptive step-length control is implemented as a module in the MATLAB package `bvpsuite2.0`.

1.2 Implementation in `bvpsuite2.0`

The main aim is the implementation of a pathfollowing module with an adaptive step-length control strategy in `bvpsuite2.0`. In this section, this implementation of the module is presented in detail.
During the implementation, the new functions `functionFDF` and `pathfollowing` were added to the package `bvpsuite2.0`. These will be presented in the form of pseudo-code and discussed in the following pages.
`bvpsuite2.0` is a numerical solver for boundary value problems (BVPs) in ordinary differential equations (ODEs). Thus, in the beginning of this section, the collocation method used in `bvpsuite2.0` is discussed. This will then be related to the results of the first section of this chapter. Afterwards, some intricacies of the implementation of the new pathfollowing module are discussed. Finally, some of its features, which may be relevant to the user, are elaborated.

1.2.1 Solution approximation in `bvpsuite2.0`

The explanations for `bvpsuite2.0` are combining parts of the work of [1] and [3]. They are revisited in this section since these play an important role in the explanations of the function `functionFDF` and thus the pathfollowing module.
First, the general type of problem `bvpsuite2.0` handles is presented. Let therefore $\mathcal{I} = [a, b] \subset \mathbb{R}$ be given, as well as a vector $\boldsymbol{c} \in \mathbb{R}^q$ with $c_i \in \mathcal{I}$ and $c_i \neq c_j$ for $i \neq j$.
Then, we try to find a vector function $\boldsymbol{z}(t) : \mathcal{I} \to \mathbb{R}^n$ and a vector of parameters $\boldsymbol{p} \in \mathbb{R}^s$, such that for all $t \in \mathcal{I}$ and for $\lambda^* \in \mathbb{R}$ the ODE system

$$\boldsymbol{f}(t, \boldsymbol{p}, z_1(t), z_1'(t), ..., z_1^{(l_1)}(t), ..., z_i(t), ..., z_k^{(l_k)}(t), ..., z_n(t), ..., z_n^{(l_n)}(t), \lambda^*) = \boldsymbol{0}, \tag{1.26}$$

and the boundary conditions

$$\boldsymbol{g}(\boldsymbol{p}, z_1(c_1), ..., z_1^{(l_1-1)}(c_1), ..., z_n(c_1), ..., z_n^{(l_n-1)}(c_1), ...$$

$$..., z_1(c_q), ..., z_1^{(l_1-1)}(c_q), ..., z_n(c_q), ..., z_n^{(l_n-1)}(c_q), \lambda^*) = \mathbf{0}, \qquad (1.27)$$

are satisfied. Here,

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

and we assume that $z_k \in C^{l_k-1}(\mathcal{I})$ and $z_k^{(l_k)}$ exist. The variable λ^* solely appears in the case of a pathfollowing problem. In any other case, it can simply be omitted entirely in the implicit equations above and also in the discussion below. As is shown below, the difference between the two cases is that in the case of pathfollowing, during the computation a scalar equation is added to the vector-valued function $\boldsymbol{f}$, which fixes the value of λ^*. The shape of this scalar equation is mentioned below and in more detail in Section 1.1.2.

In general, `bvpsuite2.0` employs a collocation method to solve the BVP (1.26)–(1.27) numerically. Therefore,

- the interval $\mathcal{I}$ is discretized by introducing the mesh $\{a = \tau_0 < \tau_1 < ... < \tau_N = b\}$ with interval lengths $h_i := \tau_i - \tau_{i-1}$ for $i = 1, ..., N$, and
- with a choice of a vector of inner collocation points $\boldsymbol{\rho} = (\rho_1, ..., \rho_j, ..., \rho_m)$, where $\rho_j \in (0, 1)$ and $\rho_i \neq \rho_j$ for $i \neq j$, the grid points in the subinterval (τ_{i-1}, τ_i) are

$$T_{ij} := \tau_{i-1} + \rho_j \, h_i, \quad i = 1, ..., N, \quad j = 1, ..., m.$$

Each component of a solution $\boldsymbol{z}$ is approximated by a piecewise polynomial continuous function, i.e. for a component z_k on the subinterval (τ_{i-1}, τ_i)

$$z_k\big|_{(\tau_{i-1},\tau_i)} \approx P_{ik} \in \mathbb{P}_{m+l_k-1}.$$

The polynomials P_{ik} are represented with the a-priori unknown l_k coefficients Y_{ikj} and the m coefficients Z_{ikj} in the Runge-Kutta basis as

$$P_{ik}(t) = \sum_{j=1}^{l_k} Y_{ikj} \Phi_{ij}(t) + h_i^{l_k} \sum_{j=1}^{m} Z_{ikj} \Psi_{ij}^{l_k} \left(\frac{t - \tau_{i-1}}{h_i} \right), \qquad (1.28)$$

where for $t \in [\tau_{i-1}, \tau_i]$,

$$\Phi_{ij}(t) := \frac{(t - \tau_{i-1})^{j-1}}{(j-1)!}$$

and

$$\Psi_{ij}^0(t) = \prod_{\substack{\nu=1 \\ \nu \neq j}}^{m} \frac{t - \rho_\nu}{\rho_j - \rho_\nu}, \quad \text{and} \quad \Psi_{ij}^k(t) = \int_{\tau_{i-1}}^{t} \Psi_{ij}^{k-1}(s) \mathrm{d}s, \quad \text{for } k > 0.$$

Then P_{ik} satisfies the relations

$$\begin{aligned} P_{ik}^{(d-1)}(\tau_{i-1}) &= Y_{ikd} \quad \text{for } d = 1, ..., l_k \quad \text{and} \\ P_{ik}^{(l_k)}(T_{ij}) &= Z_{ikj} \quad \text{for } j = 1, ..., m. \end{aligned} \tag{1.29}$$

The value of λ^* is fixed by a scalar equation – which will be specified below – in the case of a pathfollowing problem, or in any other case it can be omitted. Then, the unknown coefficients in (1.28) are defined by requiring the following three conditions:

1. the ODE system (1.26) of n equations is satisfied in all grid points, i.e. for $i = 1, ..., N$, $j = 1, ..., m$

$$\boldsymbol{f}(T_{ij}, \boldsymbol{p}, P_{i1}(T_{ij}), ..., P_{i1}^{(l_1)}(T_{ij}), ..., P_{in}(T_{ij}), ..., P_{in}^{(l_n)}(T_{ij}), \lambda_0) = \mathbf{0}, \tag{1.30}$$

2. the boundary conditions (1.27) hold, i.e.

$$\begin{aligned} &\boldsymbol{g}(\boldsymbol{p}, P_{\tilde{c}_1,1}(c_1), ..., P_{\tilde{c}_1,1}^{(l_1-1)}(c_1), ..., P_{\tilde{c}_1,n}(c_1), ..., P_{\tilde{c}_1,n}^{(l_n-1)}(c_1), ..., \\ &\quad ..., P_{\tilde{c}_q,1}(c_q), ..., P_{\tilde{c}_q,1}^{(l_1-1)}(c_q), ..., P_{\tilde{c}_q,n}(c_q), ..., P_{\tilde{c}_q,n}^{(l_n-1)}(c_q), \lambda_0) = \mathbf{0}. \end{aligned} \tag{1.31}$$

   In the index of the polynomials $P_{\tilde{c}_l,k}$, $l = 1, ..., q$, $k = 1, ..., n$, the tilde on top of c_l indicates that depending on the location of c_l the corresponding polynomial $P_{\tilde{c}_l,k}$ is chosen in such a way that $c_l \in \left[\tau_{\tilde{c}_l-1}, \tau_{\tilde{c}_l}\right)$, and

3. the piecewise polynomial function is globally continuous together with its derivatives up to the order $l_k - 1$ (for z_k), i.e. for $i = 1, ..., N-1$, $k = 1, ..., n$,

$$P_{i,k}^{(d-1)}(\tau_i) = P_{i+1,k}^{(d-1)}(\tau_i), \quad d = 1, ..., l_k. \tag{1.32}$$

   Note that by (1.29) this is equivalent to $P_{i,k}^{(d-1)}(\tau_i) = Y_{i+1,k,d}$.

The three systems of equations, namely

1. the nNm equations from (1.30),
2. the $s + \sum_{k=1}^{n} l_k$ equations from (1.31), and
3. the $(N-1)\sum_{k=1}^{n} l_k$ equations from (1.32),

define uniquely the $N\sum_{k=1}^{n} l_k$ unknown coefficients Y_{ikj}, the nNm unknown coefficients Z_{ikj} and the s unknown parameter values in the vector $\boldsymbol{p}$, provided that the mesh is fine enough.

These can be considered the building blocks of the `bvpsuite2.0` solver routine. The system of equations (1.30)–(1.32) and an equation fixing the value of λ^* is saved in a column vector $\boldsymbol{F}(\cdot) \in \mathbb{R}^{N(\sum_{k=1}^{n} l_k + nm)+s+1}$. Also its Jacobian $D\boldsymbol{F}(\cdot)$ is needed in the computation to find an approximation to $\boldsymbol{z} := (z_1, ..., z_n)$ by $\boldsymbol{P} := (P_{11}, ..., P_{1N}, ..., P_{n1}, ..., P_{nN})$. We explain the construction of the function $\boldsymbol{F}(\cdot)$, the function $D\boldsymbol{F}(\cdot)$, and their input variable $\boldsymbol{X} \in \mathbb{R}^{N(\sum_{k=1}^{n} l_k + nm)+s+1}$ below.

The input variable $\boldsymbol{X}$

Define $\hat{l} = \max(l_k)$ and set for all $k = 1, ..., n$ and $i = 1, ..., N$, $Y_{ikj} = 0$ when $l_k < j \leq \hat{l}$. Then for the i-th subinterval (τ_{i-1}, τ_i), $i = 1, ..., N$, the coefficients of the polynomials P_{ik}, $k = 1, ..., n$, are saved in the row vectors $\hat{y}_{id} := (Y_{i1d}, ..., Y_{ind})$ for $d = 1, ..., \hat{l}$ and $\hat{z}_{ij} := (Z_{i1j}, ..., Z_{inj})$ for $j = 1, ..., m$. Furthermore, the parameter input variables are saved in the row vector $\tilde{\boldsymbol{p}} = (p_1, ..., p_s)$.
Now, define for $i = 1, ..., N$

$$\boldsymbol{x}_i := (\hat{y}_{i1}, ..., \hat{y}_{i\hat{l}}, \hat{z}_{i1}, ..., \hat{z}_{im}), \qquad \text{and} \qquad \boldsymbol{x}_{N+1} := \tilde{\boldsymbol{p}}.$$

The concatenation of these $N + 1$ row vectors and the scalar value λ_0 of the variable λ^* is now denoted with

$$\boldsymbol{X} := (\boldsymbol{x}_1, ..., \boldsymbol{x}_N, \boldsymbol{x}_{N+1}, \lambda_0). \tag{1.33}$$

Thus, in $\boldsymbol{x}_i$ all the coefficients of the polynomials P_{ik}, $k = 1, ..., n$, approximating the behaviour of the solution z_k in the i-th subinterval, are saved. In $\boldsymbol{x}_{N+1}$ the s approximate parameter values are saved. And $\boldsymbol{X}$, the vector at which $\boldsymbol{F}(\cdot)$ is evaluated, contains all these values and the value λ_0 of the pathfollwoing parameter in case of pathfollowing problems.

The function $\boldsymbol{F}(\cdot)$

For a t in the i-th subinterval, the column vectors

$$\boldsymbol{P}_i^{(d)}(t) := (P_{i1}^{(d)}(t), ..., P_{in}^{(d)}(t))^T, \qquad \text{for } d = 0, ..., \hat{l},$$

are defined. The index number d represents the order of the derivative. Thus, in $\boldsymbol{P}_i^{(d)}(t)$ the values of the d-th order derivatives of the polynomials P_{ik}, $k = 1, ..., n$, evaluated at t is saved.
In the following definition of the function $\boldsymbol{F}(\cdot)$, the vectors $\boldsymbol{P}_i^{(d)}$ are used as function arguments of the BVP $\boldsymbol{f}$ and its boundary conditions $\boldsymbol{g}$. Note, that this notation is merely a rearrangement of the functions' arguments in (1.26) and (1.27).
With these definitions, we can write $\boldsymbol{F}(\cdot)$ evaluated at $\boldsymbol{X}$ as

$$\boldsymbol{F}(\boldsymbol{X}) := \begin{pmatrix} \boldsymbol{g}(\tilde{\boldsymbol{p}}, \boldsymbol{P}_{\tilde{c}_1}^{(0)}(c_1), ..., \boldsymbol{P}_{\tilde{c}_1}^{(\hat{l})}(c_1), ..., \boldsymbol{P}_{\tilde{c}_q}^{(0)}(c_q), ..., \boldsymbol{P}_{\tilde{c}_q}^{(\hat{l})}(c_q), \lambda_0) \\ \boldsymbol{f}(T_{11}, \tilde{\boldsymbol{p}}, \boldsymbol{P}_1^{(0)}(T_{11}), ..., \boldsymbol{P}_1^{(\hat{l})}(T_{11}), \lambda_0) \\ \vdots \\ \boldsymbol{f}(T_{1m}, \tilde{\boldsymbol{p}}, \boldsymbol{P}_1^{(0)}(T_{1m}), ..., \boldsymbol{P}_1^{(\hat{l})}(T_{1m}), \lambda_0) \\ \boldsymbol{P}_1^{(0)}(\tau_1) - \hat{y}_{21}^T \\ \boldsymbol{P}_1^{(1)}(\tau_1) - \hat{y}_{22}^T \\ \vdots \\ \boldsymbol{P}_1^{(\hat{l}-1)}(\tau_1) - \hat{y}_{2,\hat{l}}^T \\ \boldsymbol{f}(T_{21}, \tilde{\boldsymbol{p}}, \boldsymbol{P}_2^{(0)}(T_{21}), ..., \boldsymbol{P}_2^{(\hat{l})}(T_{21}), \lambda_0) \\ \vdots \\ \boldsymbol{f}(T_{2m}, \tilde{\boldsymbol{p}}, \boldsymbol{P}_2^{(0)}(T_{2m}), ..., \boldsymbol{P}_2^{(\hat{l})}(T_{2m}), \lambda_0) \\ \boldsymbol{P}_2^{(0)}(\tau_2) - \hat{y}_{31}^T \\ \boldsymbol{P}_2^{(1)}(\tau_2) - \hat{y}_{32}^T \\ \vdots \\ \boldsymbol{P}_2^{(\hat{l}-1)}(\tau_2) - \hat{y}_{3,\hat{l}}^T \\ \vdots \\ \boldsymbol{f}(T_{Nm}, \tilde{\boldsymbol{p}}, \boldsymbol{P}_N^{(0)}(T_{Nm}), ..., \boldsymbol{P}_N^{(\hat{l})}(T_{Nm}), \lambda_0) \\ t^*(\boldsymbol{X}) \end{pmatrix}. \tag{1.34}$$

As previously mentioned, the vector $\boldsymbol{F}$ contains the conditions (1.30)–(1.32) in its definition and an equation $t^*(\boldsymbol{X})$ fixing the value of λ^*. The equation has one of two shapes, i.e.

1. when computing the tangent to the homotopy path in a point $\boldsymbol{y}_0 := (\boldsymbol{x}_1, ..., \boldsymbol{x}_{N+1}, \lambda_0)$ during the prediction step

$$t^*(\boldsymbol{X}) := \lambda^* - \lambda_0; \tag{1.35}$$

2. when computing the function $\boldsymbol{F}(\cdot)$ and $\boldsymbol{DF}(\cdot)$ for the Newton iteration in the correction step, first the equation

$$\boldsymbol{DF}(\boldsymbol{X})\boldsymbol{t} = (0, ..., 0, 1)^T$$

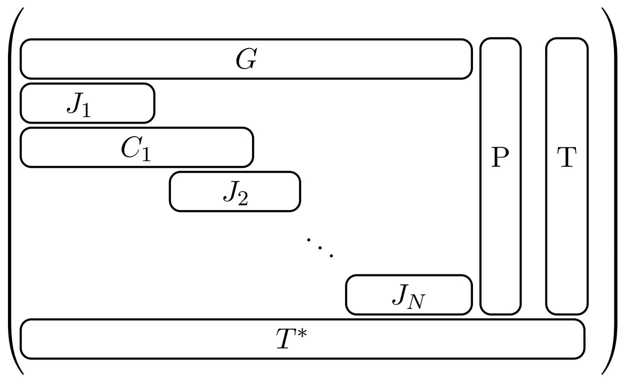
is solved for $\boldsymbol{t}$. For this equation, $\boldsymbol{DF}(\boldsymbol{X})$ is computed as described in the case above when computing the tangent. After computing the tangent $\boldsymbol{t}$, it is normalized to $\boldsymbol{t}_0$, thus $\|\boldsymbol{t}_0\| := 1$. Finally, for a previously computed solution vector $\boldsymbol{y}_0$ and the step-length s, the equation fixing the value of the variable λ^* is

$$t^*(\boldsymbol{X}) := (\boldsymbol{X} - (\boldsymbol{y}_0 + s \cdot \boldsymbol{t}_0)) - s. \tag{1.36}$$

The function $\boldsymbol{DF}(\cdot)$

The Jacobian of $\boldsymbol{F}(\cdot)$, $\boldsymbol{DF}(\cdot)$, evaluated at $\boldsymbol{X}$ has the block-structure

$\boldsymbol{DF}(\boldsymbol{X}) =$



. (1.37)

The blocks of the Jacobian matrix $\boldsymbol{DF}$ are

- the block G, which contains the derivatives of the $s + \sum_{k=1}^{n} l_k$ first rows of $\boldsymbol{F}(\cdot)$ with respect to $\boldsymbol{x}_i$, $i = 1, ..., N$, evaluated at $\boldsymbol{X}$. Then, G_u, the u-th row of G, can be written as

$$G_u := \left(\frac{\partial \boldsymbol{g}_u}{\partial \hat{y}_{11}}, ..., \frac{\partial \boldsymbol{g}_u}{\partial \hat{y}_{1,\hat{l}}}, \frac{\partial \boldsymbol{g}_u}{\partial \hat{z}_{11}}, ..., \frac{\partial \boldsymbol{g}_u}{\partial \hat{z}_{1m}}, \frac{\partial \boldsymbol{g}_u}{\partial \hat{y}_{21}}, ..., \frac{\partial \boldsymbol{g}_u}{\partial \hat{z}_{Nm}} \right),$$

  where $\boldsymbol{g}_u$ is the u-th row of $\boldsymbol{g}$. We now define for $k = 1, ..., n$, $\alpha = 1, ..., q$, $r = 1, ..., \hat{l}$,

$$D_{k\alpha r}\boldsymbol{g}_u := \left. \frac{\partial \boldsymbol{g}_u}{\partial z_k^{(r)}} \right|_{t=c_\alpha, \tilde{\boldsymbol{p}}=\boldsymbol{p}, \boldsymbol{z}=\boldsymbol{P}, \lambda^*=\lambda_0}, \tag{1.38}$$

  where $D_{k\alpha r}\boldsymbol{g}_u := 0$ for $r \geq l_k$. Then, with the chain rule, we have

$$\frac{\partial \boldsymbol{g}_u}{\partial Y_{ikj}} = \sum_{\{\alpha | \tilde{c}_\alpha = i\}} \sum_{r=0}^{j-1} \left. \frac{\partial \boldsymbol{g}_u}{\partial z_k^{(r)}} \right|_{t=c_\alpha, \tilde{\boldsymbol{p}}=\boldsymbol{p}, \boldsymbol{z}=\boldsymbol{P}, \lambda^*=\lambda_0} \cdot \frac{\partial P_{ik}^{(r)}}{\partial Y_{ikj}}(c_\alpha)$$

$$= \sum_{\{\alpha|\tilde{c}_\alpha=i\}} \sum_{r=0}^{j-1} D_{k\alpha r}\boldsymbol{g}_u \cdot \frac{(c_\alpha - \tau_{i-1})^{(j-1-r)}}{(j-1-r)},$$

$$\frac{\partial \boldsymbol{g}_u}{\partial Z_{ikj}} = \sum_{\{\alpha|\tilde{c}_\alpha=i\}} \sum_{r=0}^{l_k} \left.\frac{\partial \boldsymbol{g}_u}{\partial z_k^{(r)}}\right|_{t=c_\alpha,\tilde{\boldsymbol{p}}=\boldsymbol{p},\boldsymbol{z}=\boldsymbol{P},\lambda^*=\lambda_0} \cdot \frac{\partial P_{ik}^{(r)}}{\partial Z_{ikj}}(c_\alpha)$$

$$= \sum_{\{\alpha|\tilde{c}_\alpha=i\}} \sum_{r=0}^{l_k} D_{k\alpha r}\boldsymbol{g}_u \cdot \Psi_{ij}^{l_k-r}(c_\alpha),$$

since

$$\frac{\partial P_{ik}^{(r)}}{\partial Y_{ikj}}(t) = \Phi_{ij}^{(r)}(t) = \Phi_{i,(j-r)}(t), \text{ and}$$
$$\frac{\partial P_{ik}^{(r)}}{\partial Z_{ikj}}(t) = (\Psi_{ij}^{l_k})^{(r)}(t) = \Psi_{ij}^{l_k-r}(t).$$

- the blocks J_i, where for $i = 1, ..., N$ and $j = 1, ..., m$

$$f_{ij} := \boldsymbol{f}(T_{ij}, \tilde{\boldsymbol{p}}, \boldsymbol{P}_1^{(0)}(T_{ij}), ..., \boldsymbol{P}_1^{(\hat{l})}(T_{ij}))$$

is defined. Then, the block J_i, with $i \in \{1, ..., N\}$, contains the m rows of derivatives of the n-rows vector f_{ij}, $j = 1, ..., m$, with respect to $\boldsymbol{x}_i$, i.e.

$$J_i := \begin{pmatrix} \frac{\partial f_{i1}}{\partial \hat{y}_{i1}} & \cdots & \frac{\partial f_{i1}}{\partial \hat{y}_{i,\hat{l}}} & \frac{\partial f_{i1}}{\partial \hat{z}_{i1}} & \cdots & \frac{\partial f_{i1}}{\partial \hat{z}_{im}} \\ \frac{\partial f_{i2}}{\partial \hat{y}_{i1}} & \cdots & \frac{\partial f_{i2}}{\partial \hat{y}_{i,\hat{l}}} & \frac{\partial f_{i2}}{\partial \hat{z}_{i1}} & \cdots & \frac{\partial f_{i2}}{\partial \hat{z}_{im}} \\ \vdots & & \vdots & \vdots & & \vdots \\ \frac{\partial f_{im}}{\partial \hat{y}_{i1}} & \cdots & \frac{\partial f_{im}}{\partial \hat{y}_{i,\hat{l}}} & \frac{\partial f_{im}}{\partial \hat{z}_{i1}} & \cdots & \frac{\partial f_{im}}{\partial \hat{z}_{im}} \end{pmatrix}.$$

Analogously as for the block G, for the u-th row of f_{ij}, we define for $i = 1, ..., N$, $j = 1, ..., m$, $k = 1, ..., n$, $r = 1, ..., \hat{l}$,

$$D_{kr} f_{ij;u} := \left.\frac{\partial \boldsymbol{f}_u}{\partial z_k^{(r)}}\right|_{t=T_{ij},\tilde{\boldsymbol{p}}=\boldsymbol{p},\boldsymbol{z}(t)=\boldsymbol{P}(T_{ij}),\lambda^*=\lambda_0}, \tag{1.39}$$

where $D_{k\alpha r}\boldsymbol{g}_u := 0$ for $r \geq l_k$ and $\boldsymbol{f}_u$ is the u-th row of $\boldsymbol{f}$. Then again, with the chain rule, we have

$$\frac{\partial f_{ij;u}}{\partial Y_{ikj}} = \sum_{r=0}^{j-1} \left.\frac{\partial \boldsymbol{f}_u}{\partial z_k^{(r)}}\right|_{t=T_{ij},\tilde{\boldsymbol{p}}=\boldsymbol{p},\boldsymbol{z}(t)=\boldsymbol{P}(T_{ij}),\lambda^*=\lambda_0} \cdot \frac{\partial P_{ik}^{(r)}}{\partial Y_{ikj}}(T_{ij})$$

$$= \sum_{r=0}^{j-1} D_{kr} f_{ij;u} \cdot \frac{(T_{ij} - \tau_{i-1})^{(j-1-r)}}{(j-1-r)},$$

$$\begin{aligned}\frac{\partial f_{ij;u}}{\partial Z_{ikj}} &= \sum_{r=0}^{l_k} \left.\frac{\partial \boldsymbol{f}_u}{\partial z_k^{(r)}}\right|_{t=T_{ij},\tilde{\boldsymbol{p}}=\boldsymbol{p},\boldsymbol{z}(t)=\boldsymbol{P}(T_{ij}),\lambda^*=\lambda_0} \cdot \frac{\partial P_{ik}^{(r)}}{\partial Z_{ikj}}(T_{ij}) \\ &= \sum_{r=0}^{l_k} D_{kr} f_{ij;u} \cdot \Psi_{ij}^{l_k-r}(T_ij).\end{aligned}$$

- the blocks C_i, where $i \in \{1, ..., N-1\}$, which contains the derivatives of the vectors $\boldsymbol{P}_i^{(r)} - \hat{y}_{(i+1),r+1}$, $r = 0, ..., \hat{l}-1$, with respect to $\boldsymbol{x}_i$ and $(\hat{y}_{(i+1),1}, ..., \hat{y}_{(i+1),\hat{l}})$, evaluated at τ_i, i.e.

$$C_i := \begin{pmatrix} \frac{\partial \boldsymbol{P}_i}{\partial \hat{y}_{i1}}(\tau_i) & ... & \frac{\partial \boldsymbol{P}_i}{\partial \hat{y}_{i,\hat{l}}}(\tau_i) & \frac{\partial \boldsymbol{P}_i}{\partial \hat{z}_{i1}}(\tau_i) & ... & \frac{\partial \boldsymbol{P}_i}{\partial \hat{z}_{im}}(\tau_i) & -\frac{\partial \hat{y}^T_{(i+1),1}}{\partial \hat{y}_{(i+1),1}} & \\ \vdots & & & & & & \ddots & \\ \frac{\partial \boldsymbol{P}_i^{(\hat{l})}}{\partial \hat{y}_{i1}}(\tau_i) & ... & \frac{\partial \boldsymbol{P}_i^{(\hat{l})}}{\partial \hat{y}_{i,\hat{l}}}(\tau_i) & \frac{\partial \boldsymbol{P}_i^{(\hat{l})}}{\partial \hat{z}_{i1}}(\tau_i) & ... & \frac{\partial \boldsymbol{P}_i^{(\hat{l})}}{\partial \hat{z}_{im}}(\tau_i) & & -\frac{\partial \hat{y}^T_{(i+1),1}}{\partial \hat{y}_{(i+1),\hat{l}}} \end{pmatrix}$$

  The entries of the blocks C_i are computed in the same way as presented above for the entries of the blocks G and J_i.
- the block P, which contains the derivatives of all entries of $\boldsymbol{F}(\cdot)$ with respect to $\tilde{\boldsymbol{p}}$, i.e.

$$P := \begin{pmatrix} \frac{\partial \boldsymbol{F}_1}{\partial p_1} & ... & \frac{\partial \boldsymbol{F}_1}{\partial p_s} \\ \vdots & \ddots & \vdots \\ \frac{\partial \boldsymbol{F}_{N(nm+\sum l_i)+s}}{\partial p_1} & ... & \frac{\partial \boldsymbol{F}_{N(nm+\sum l_i)+s}}{\partial p_s} \end{pmatrix}, \tag{1.40}$$

  where $\boldsymbol{F}_u$ with $u = 1, ..., N(nm + \sum l_i) + s$ denotes the u-th row of $\boldsymbol{F}(\cdot)$.
- the block T, which contains the derivatives of all entries of $\boldsymbol{F}(\cdot)$ with respect to λ^*, i.e.

$$T := \left(\frac{\partial \boldsymbol{F}_1}{\partial \lambda^*}, ..., \frac{\partial \boldsymbol{F}_{N(nm+\sum l_i)+s}}{\partial \lambda^*}\right)^T, \tag{1.41}$$

  where $\boldsymbol{F}_u$ with $u = 1, ..., N(nm + \sum l_i) + s$ denotes the u-th row of $\boldsymbol{F}(\cdot)$.
- the block T^*, which contains the derivatives of $t^*(\cdot)$ with respect to $\boldsymbol{X}$, i.e.

$$T^* := \left(\frac{\partial t^*}{\partial \hat{y}_{11}}, ..., \frac{\partial t^*}{\partial \hat{z}_{Nm}}, \frac{\partial t^*}{\partial p_1}, ..., \frac{\partial t^*}{\partial p_s}, \frac{\partial t^*}{\partial \lambda^*}\right). \tag{1.42}$$

  In the two forms of $t^*(\cdot)$ described above, the result is

  1. when computing the Jacobian for the solution of the tangent equation in the prediction step

$$T^* = (0, ..., 0, 1)^T;$$

2. when computing the Jacobian for the Newton iteration in the correction step

$$T^* = \boldsymbol{t}_0.$$

In order for `bvpsuite2.0` to be able to run its computations, the derivatives presented in (1.38), (1.39), (1.40), (1.41) and (1.42) need to be provided by the user in the problem definition. These definitions are then called in the routine of `bvpsuite2.0`.

If the implicit problem $\boldsymbol{f} = \mathbf{0}$ with boundary conditions $\boldsymbol{g} = \mathbf{0}$, given as in (1.26) and (1.27), is linear, then the system of linear equations $D\boldsymbol{F}(\mathbf{0})\boldsymbol{x} = \boldsymbol{F}(\mathbf{0})$ has to be solved for $\boldsymbol{x}$, where $\boldsymbol{F}$ is as in (1.34), $D\boldsymbol{F}$ is as in (1.37) and $\boldsymbol{x}$ is the solution vector in the same format as in (1.33).

If the implicit problem $\boldsymbol{f} = \mathbf{0}$ with boundary conditions $\boldsymbol{g} = \mathbf{0}$ is non-linear, then an initial guess $\boldsymbol{x}_0$ to approximate the solution is taken and the fast frozen Newton method is applied to it. The idea behind the Fast Frozen Newton method is briefly mentioned in Section 1.1.2 and in more detail in [2].

Depending on the argument of the function call, the function `functionFDF` returns the vector $\boldsymbol{F}(\boldsymbol{X}_0)$ or the square matrix $D\boldsymbol{F}(\boldsymbol{X}_0)$ for a given $\boldsymbol{X}_0$, which is also an argument of the function call. In Algorithm 1, the basic structure of the function `functionFDF` is presented.

Algorithm 1: `functionFDF.m`: Computation of $\boldsymbol{F}(\boldsymbol{X}_0)$ and $D\boldsymbol{F}(\boldsymbol{X}_0)$.

```
Input: Request and input vector $\boldsymbol{X}_0$
Output: The vector $\boldsymbol{F}(\boldsymbol{X}_0)$ or the square matrix $D\boldsymbol{F}(\boldsymbol{X}_0)$
                                                                    // In functionFDF.m at line
1  switch Request do                                                              // 3
2      case $\boldsymbol{F}(\boldsymbol{X}_0)$ do                                 // 4
3          Extract polynomial coefficients and parameters from $\boldsymbol{X}_0$ ;    // 33
4          Initialize output vector, same size as $\boldsymbol{X}_0$ ;           // 53
5          Boundary conditions are evaluated ;                                    // 55
6          Problem equations are evaluated ;                                      // 92
7          if Pathfollowing then $t^*(\boldsymbol{X}_0)$ as in (1.35) or (1.36) is evaluated,
             depending on whether tangent or Newton method is needed;             // 124
8      case $D\boldsymbol{F}(\boldsymbol{X}_0)$ do                                // 128
9          Extract polynomial coefficients and parameters from $\boldsymbol{X}_0$ ;    // 163
10         Initialize sparse, square output matrix with $N(nm + \sum l_i) + s + 1$
             rows ;                                                               // 53
11         Block $G$ and upper part of $P$ are computed ;                         // 198
12         Blocks $J_i$ and $C_i$ and lower part of $P$ are computed ;            // 418
13         if Pathfollowing then                                                  // 563
14             Block $T$ is computed ;                                            // 565
15             Block $T^*$ is computed ;                                          // 641
16         end
17     end
18 end
```

This concludes the discussion of the core functionality of `bvpsuite2.0`. Having explained the auxiliary function `functionFDF`, it is now time to present the main function of the pathfollowing module.

1.2.2 Details of the implementation

In this subsection, some details of the implementation of the function `pathfollowing.m` are presented. A flowchart map is provided in Figure 1.2. Clearly, if the problem specification is associated with a pathfollowing problem, the code enters the function `pathfollowing`. Only once the user chooses to save the results and exit the program, the code will return to the main function `bvpsuite2`. There the computed data is saved and the program terminated.
This means that the computations begin in the problem definition file, where the user specifies the problem for the code to process. Before the code can be started, the solver settings need to be adjusted. Once this is completed, the pathfollowing can start. The explanations in the following pages will proceed in this chronological order.

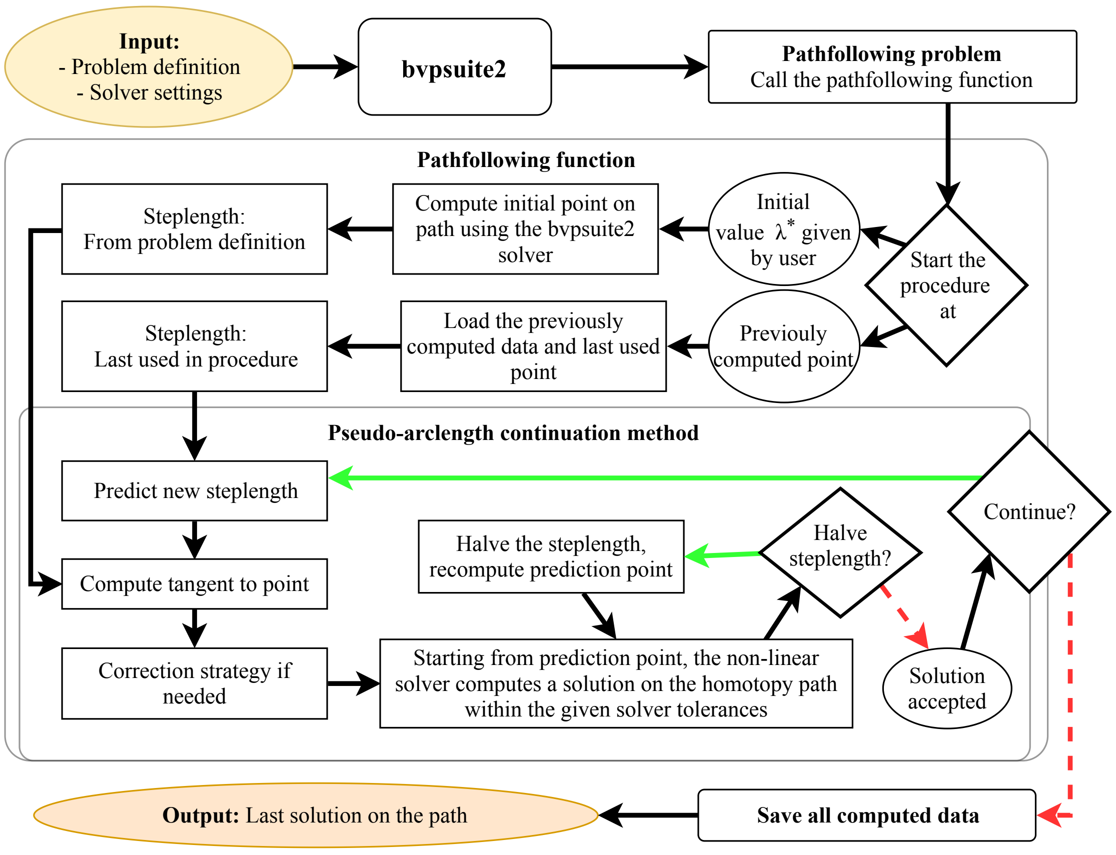

Input:
- Problem definition
- Solver settings
bvpsuite2
Pathfollowing problem
Call the pathfollowing function
Pathfollowing function
Steplength: From problem definition
Compute initial point on path using the bvpsuite2 solver
Initial value λ* given by user
Start the procedure at
Steplength: Last used in procedure
Load the previously computed data and last used point
Previouly computed point
Pseudo-arclength continuation method
Predict new steplength
Continue?
Halve the steplength, recompute prediction point
Halve steplength?
Compute tangent to point
Correction strategy if needed
Starting from prediction point, the non-linear solver computes a solution on the homotopy path within the given solver tolerances
Solution accepted
Output: Last solution on the path
Save all computed data

Figure 1.2: Flowchart of the pathfollowing function.

Changes to the problem definition file

The problems the pathfollowing module is suited to deal with are BVP equations of the same form as in (1.26)–(1.27) with a parameter λ^*, which is varied along a path. To pass some key values of the pathfollowing procedure to `bvpsuite2.0`, a specific case was added in the `switch`-block in the problem definition file. This is the `case 'pathfollowing'` which will return a MATLAB-structure variable, containing the fields

- `activate`: boolean, if `1` then the problem is a pathfollowing problem, otherwise `0`;
- `pathdata`: handle to the function `PathCharData`, computing the functional of the solution function that is being followed;
- `startat`: either `'start'` or the name of a file contained in the folder specified under `dir`;
- `start`: first value of the pathfollowing parameter taken during the run, this is only considered if `startat` is set to `'start'`;
- `steplength`: first value of the step-length taken during the run, this is only considered if `startat` is set to `'start'`;

- `max_pred_length`: (*can be omitted*) maximal length of the predictor step;
- `pit_stop`: (*can be omitted*) a row vector containing 2 entries, a first value when reached by the pathfollowing parameter, the user will be prompted how to proceed, and a second value when reached by the value of the function `PathCharData` during the run, the user will be also prompted how to proceed;
- `require_exact`: (*can be omitted*) a row vector of values when reached by the pathfollowing variable, the solution of the BVP at this specific pathfollowing parameter value shall be computed and saved in the MATLAB-cell variable `speicher_exact`, which is one of the 3 outputs of the function `pathfollowing`;
- `dir`: the folder in which the data is saved or from where it may be loaded, and
- `name`: the name under which the path data consisting of `speicher`, `speicher_exact` and `tur_pts`, is saved;
- `counter`: (*can be omitted*) the number of pathfollowing steps that are executed before the user is prompted how to proceed. This can also be set to `Inf` or a very high number, then the code will run until a value from `pit_stop` is reached. This option is only recommended for already tested problems, since some undesired issues may occur along the way otherwise;
- `only_counter`: (*can be omitted*) this is a boolean value used with `counter`. If it is set to `1`, then the number of steps given in `counter` will be performed and then the path automatically saved without any further user input. If it is set to `0`, nothing will happen;
- `only_exact`: (*can be omitted*) this is used with `require_exact` and `startat`. Starting at a previously computed path, a single step is performed and afterwards all the approximations to the solutions of the equations with the pathfollowing parameter being equal to the values in the vector `require_exact`, are computed.

The aim of the whole procedure is to follow a characteristic value of the solution of the problem along the variation of the pathfollowing parameter. The function `PathCharData` responsible for computing this characteristic value needs to be given by the user in the problem definition file. It requires the inputs

1. `x1`: contains the current mesh,
2. `coeff`: a vector, in the shape of (1.33), composed of

   - the coefficients of the polynomials in the Runge-Kutta basis approximating the solutions at the current point,
   - the parameters in case of a parameter-dependent problem,
   - the approximation to the eigenvalue in case of an eigenvalue problem, and
   - the current pathfollowing parameters value,

   which all fit the current mesh,
3. `ordnung`: the orders of the unknown functions of the problem, and
4. `rho`: a vector of the collocation points.

In summary, the whole input data is chosen to fit the input data into the `coeffToValues`

function, which returns from a given vector of coefficients in the Runge-Kutta basis (1.28) the approximated values of the function or one of its derivatives at the requested points. See Section 2.1 for details on the function. The column vector `ret` is returned as output by the function `PathCharData`.
With `PathCharData` it is possible to

- evaluate a solution function or its derivative at a specific point. The lines of code for evaluating $z_1(0)$ for instance are

  ```
  help=coeffToValues( coeff, x1,ordnung,rho,0, 0);
  ret = help(1);
  ```

  and the lines of code for evaluating $z_7''(15)$ are

  ```
  help=coeffToValues( coeff, x1,ordnung,rho,15, 2);
  ret = help(7);
  ```

  In case of a problem which is posed on $[0, \infty)$, the problem is transformed to a problem on $[0, 1]$, as briefly explained in Section 2.1, or in more detail in [1, Chap. 3]. The number of unknown functions n is doubled to $2n$ during the transformation. Thus `coeffToValues` when evaluated at 0.5 for instance, returns a vector or matrix with $2n$ rows, the first n rows corresponding to the values of the n solution functions at 0.25 and the second n rows corresponding to the values of the n solution functions at $\frac{1}{0.25} = 4$. Therefore, in order to evaluate, as above, $z_7''(15)$ for a problem with 18 unknown functions, posed on a semi-infinite interval, the lines of code are

  ```
  help=coeffToValues( coeff, x1,ordnung,rho, 1/15, 2);
  ret = help(18+7);
  ```

  This is due to the fact, that the transformation $t \mapsto \frac{1}{t}$ for $t > 1$ is used to transform the problem on $[0, \infty)$ to a problem on $[0, 1]$, thus augmenting the original equations in $[0, 1]$.

- return the value of the parameter at the given point. For an example with 15 parameters in the problem, where the evolution of the fifth parameter is to be followed, then this is realized by the line

  ```
  ret = coeff(end-1-15+5);
  ```

  This is due to the fact, that at the end of the `coeff` vector, the pathfollowing parameter value is stored last and before that the 15 approximate values of the parameters at this stage of the process. For an other example with three parameters in the problem, where the evolution of the first parameter is to be followed, then this is realized by the line

  ```
  ret = coeff(end-1-3+1);
  ```

  When dealing with an eigenvalue problem, then the evolution of the eigenvalue can be followed by defining the return value of the function `PathCharData` as

```
ret = coeff(end-1);
```

The approximation to the eigenvalue is saved in the penultimate step of the `coeff` vector. Actually the eigenvalue is treated as the $p+1$-th parameter, more details on the treatment of eigenvalue problems in `bvpsuite2.0` can be found in [1, Chap. 4].

- apply a user-defined norm to the solution function or one of its derivatives. For instance, for a problem defined on the interval $[-2,3]$ with three unknown functions, when using the L^∞-norm on the solution function z_2, this can be done by computing

  ```
  help=coeffToValues( coeff, x1,ordnung,rho,-2:1/1000:3, 0);
  ret = max(abs(help(2,:));
  ```

  For another problem, defined on the interval $[-6,-5]$ with 8 unknown functions, when using the Euclidean norm on the third derivative of the solution function z_5, this can be done by computing

  ```
  help=coeffToValues( coeff, x1,ordnung,rho,-6:1/1000:-5, 3);
  ret = sqrt(help(5,:)'*help(5,:));
  ```

These are some of commonly followed characteristic values in pathfollowing problems. It is also possible to follow more than one of these values at once in the code, namely by defining the `ret` output variable as a column vector instead of a scalar value.
In the problem definition file, the two cases `'path_jac'`, which covers the derivative of the problem equations with respect to the pathfollowing variable λ^*, and `'path_dBV'`, which covers the derivative of the boundary conditions with respect to λ^*, were added, in order to compute the Jacobian as described in Section 1.2.1.

Changes to the solver settings file

The user can choose to solve the problem without any error estimation or mesh adaptation performed by `bvpsuite2.0`, with only error estimation or with mesh adaptation enabled until the error tolerances are met by the approximations. These are all set in the solver settings and performed during the pathfollowing. The differences in the treatment of these cases is explained within the explanations of the main function `pathfollowing`.
The correction and prediction strategy are as presented in Section 1.1.4. The choice of $\theta_{\max} \leq \frac{1}{4}$ is left for the user to be set in the solver settings under `'thetaMax'`, since this choice heavily influences the evolution of the step-length. As can be seen in (1.25), when $\theta_{\max}$ is chosen close to $\frac{1}{4}$, the step-length might grow more quickly and only little adjustment around critical points may be observed. On the other hand, when $\theta_{\max}$ is chosen to be a few powers of 10 smaller, then a more distinctive step-length adaptation behaviour may be observed around turning points. The growth of the predicted step-length compared to the one used in the previous step is limited by the value prescribed by the user in the solver settings under `'maxSteplengthGrowth'`. The prediction and correction strategies for the step-length are both performed in the local function `PredCorrStrat`.
Alongside the step-length control strategy, a few other cautionary measures were implemented in the code. These will always lead to halving the step-length after a step was

computed and the conditions are not satisfied, and the step is then recomputed. These cautionary measures are halving the step-length to recompute the current step, when

1. the trust region method is activated. This method is turned off during the pathfollowing procedure;
2. the cosine of the angle between the current step and the tangent in the next step is lower than the value the user prescribed in the solver settings under `'angleMin'`; this setting can be turned off by choosing the value `-1`;
3. the suggested number of new mesh points is higher than the number of mesh points at the beginning of the step multiplied with the value the user prescribed in the solver settings under `'meshFactorMax'`;
4. the distance between the predictor point and the corrected point on the homotopy path is – compared to the distance between the starting point and the predictor point – too long, according to the value the user prescribed in the solver settings under `'PredLengthFactor'`; this setting can be turned off by choosing the value `0`;
5. the distance between the predictor point and the corrected point on the homotopy path is – compared to this distance in the previous step – too long, according to the value the user prescribed in the solver settings under `'CorrLengthGrowth'`; this setting can be turned off by choosing the value `Inf`.

These measures proved to be useful when following along the homotopy path for certain examples. If the step-length is halved for any of those 5 reasons, then for the prediction of the step-length, which is used in the next step, the number in `'maxSteplengthGrowth'` is divided either by $\sqrt{2}$ to the power the number of times the step-length was halved in the previous step, or if this number is smaller than 1, the step-length is retained. Furthermore, at the beginning of each loop iteration, if the step-length gets smaller than the value `min_sl`, which is defined at line `64`, either through step-halving or during the prediction-correction procedure, then the step-length is augmented to the value `min_sl` and the iteration stops to wait for user-input on how to continue, as it is described in Section 1.2.3.

The distances between the points mentioned above and also the angle, are computed by means of the values of λ^* and the values returned by `PathCharData` at the predictor and corrector points. Distances are simply computed by the Euclidean metric in the two-dimensional space (λ^*, x), where x is the return value of `PathCharData`, and the angles with the Euclidean dot product and norm. The vectors of points consist then of the value of the parameter in one dimension and the value of the characteristic value of the solution that is being followed in the second dimension. This leads to straightforward computations, when the function responsible for the computation of the characteristic values is set correctly, as shown in the examples for `PathCharData` above. On the other hand, relying so much on the data that is being plotted for a single characteristic value of the solution of a boundary value problem, may lead to some unexpected difficulties, for which the option remains for the user to turn these safety measures off, by choosing the values mentioned in the list above in the solver settings file.

Altogether, in the solver settings the following cases were added to the `switch`-block:

```
case 'thetaMax' % Controls first Newton contraction factor, must be smaller
     than 0.25
        ret=0.1;
case 'maxCorrSteps' % Maximal # of times the predicted step-length gets
    corrected (multiplied by (1/2)^(1/2)), if the first Newton contraction
    factor is not smaller than theta_max
        ret=5; % Setting to disable: ret=Inf;
case 'maxSteplengthGrowth' % Maximal growth of the step-length when
    predicting it for the next step
        ret=2; % Setting to disable: ret=Inf;
case 'angleMin' % Minimally allowed value of cosine of angle between
    current step and tangent in the next step
        ret=0.75; % Setting to disable: ret=-1;
case 'meshFactorMax' % # of mesh points can be augmented to maximally ret-
    times the # of mesh points at the beginning of step during a single
    pathfollowing step
        ret=2; % Setting to disable: ret=Inf;
case 'PredLengthFactor' % Corrector step is maximally allowed to be (1/ret)
    -times as long as the predictor step
        ret=2; % Setting to disable: ret=0;
case 'CorrLengthGrowth' % Corrector step is maximally allowed to be ret-
    times as long as the previous corrector step
        ret=8; % Setting to disable: ret=Inf;
```

The values above are chosen more or less at random. The values were slightly adjusted in most of the problems presented in Section 1.3. Also, the values to make the restrictions void are specified.

The main function `pathfollowing`

The pseudo-code for the function `pathfollowing` is presented in Algorithm 2.
The cases with or without error estimation do not need any specific intermediate steps, the code loops over one point on the homotopy path to the next point as described in Section 1.1.2. On the other hand, the case with enabled mesh adaptation, requires special precautions. The tangent continuation procedure starts from a point $\boldsymbol{a}_0$ on the homotopy path, approximated by $\boldsymbol{X}_0$ on a discrete mesh $\mathcal{M}_0$, and returns $\boldsymbol{X}_1$, which approximates another point $\boldsymbol{a}_1$ on the homotopy path, on a discrete mesh $\mathcal{M}_1$. When the mesh $\mathcal{M}_1$ is different from $\mathcal{M}_0$, there may be more points in one mesh than the other or the points were displaced in one of the meshes, thus the vectors $\boldsymbol{X}_0$ and $\boldsymbol{X}_1$ are not defined on the same mesh. This also means that the tangent vector $\boldsymbol{t}_0$ in $\boldsymbol{X}_0$ is not defined on the same mesh as $\boldsymbol{t}_1$, the tangent vector in $\boldsymbol{X}_1$. This is a problem in the application of the formula (1.25) to predict the next step-length or when the angle between the current and the next tangent is computed. This problem is circumvented by transforming the tangent $\boldsymbol{t}_0$ from the mesh $\mathcal{M}_0$ to its equivalent $\boldsymbol{t}_1$ on the mesh $\mathcal{M}_1$.

This is done using the following lines of code:

```
initP.initialMesh=x1; % x1 is the new mesh
initP.parameters = sol.parameters; % if the problem is parameter dependent
initP.initialValues = coeffToValues(tangent_new, predictor.x1,ordnung,rho,
    x1); % evaluate the polynomials with the coefficients from tangent_new
    defined on the old mesh predictor.x1 at the new mesh points x1
initP = initial_coefficients(problem,x1,initP,rho,0); % adds a field
    containing the coefficients on the new mesh x1 to the struct initP
tangent_new = [ initP.initialCoeff ; tangent_new(end) ] ; % the tangent
    vector is updated
```

The same scheme is also used in the same way to transform the initial point of the current step $\boldsymbol{a}_0$, approximated by $\boldsymbol{X}_0$, which is the vector containing all coefficients, parameter values and pathfollowing parameter value, to fit the new mesh. This method is also used in the functions `errorestimate` and `meshadaptation`, where the meshes are adapted. The workings of the functions `initial_coefficients` and `coeffToValues` are explained in Section 2.1 and there it is also shown at which point this scheme was applied.

Algorithm 2: `pathfollowing.m`: pathfollowing with automatic step-length control strategy

```
  Input: Problem definition, solver settings, pathfollowing data
  Output: speicher, speicher_exact and tur_pts
                                          // In pathfollowing.m at line
1  Set and prepare options ;                                        // 0004
2  Initialize ;                                                     // 0016
3  if start procedure at starting value then                        // 0258
4      Initialize the needed variables;
5  else if start procedure at previously computed path then         // 0304
6      Load data ;
7      Prepare predictor point for the next step ;                  // 0387
8  end
9  while 1 do                                                       // 0417
10     Reduce step-length to minimal allowed step-length if needed ; // 0421
11     bvpsuite2.0 routine for non-linear problems ;                // 0475
12     if Initial solution was computed in this step then go to the end of the
         while loop ;                                               // 0524
13     Compute everything needed for the cautionary measures ;      // 0524
14     if Step-length needs halving then                            // 0601
15         Halve step-length and go to the end of the while loop ;
16     else                                                         // 0615
17         Save computed path data to speicher ;                    // 0630
18         Compute solution at given point if passed and save to
             speicher_exact ;                                       // 0641
19         Step itnum is completed ;                                // 0886
20     end
21     if jj== #steps in one go then                                // 0951
22         Plot if enabled ;                                        // 0954
23         Prompt user to go back to previous checkpoint if enabled ; // 1041
24         Prompt user to continue or save the data & return ;      // 1131
25         jj= 0 ;
26     end
27     Call local function PredCorrStrat to prepare next step ;     // 1225
28     jj=jj+1 ;                                                    // 1232
29 end
```

The output of the `pathfollowing` function are all passed to the main function `bvpsuite2` . There they are saved in the folder with the name, as requested by the user. This output consists of

- the MATLAB-cell variable `speicher`, containing all computed path data,
- the MATLAB-cell variable `speicher_exact`, containing data corresponding to the user-

given values, and

- the 3-rows matrix `tur_pts`, containing, for each turning point, the value of λ^* in the step before the turning point, in the step at the turning point and in the step after the turning point. This means that for each column the value in the second row is either smaller than the values in the first and third row, or it is greater than the values in the first and third row. If turning points were not encountered in the run, then `tur_pts` is an empty matrix.

These saved values can be loaded to be examined by the user. The cell-variables `speicher` and `speicher_exact` both have the same structure, that is, in the first cell line, they contain the solution struct of the point on the path, for λ^* as specified in the second cell line. Then in the third and final cell line, the values of the characteristic values which were followed along the path, computed by `PathCharData`, at the point λ^* as given in the second cell line, is saved.

Some further features were built in to the module for the convenience of the user. These are presented in the following pages.

1.2.3 Features of the pathfollowing module

In this section some details are given of a few features of the implementation.

Before the computations start, the user can choose a maximal length of the predictor step. If the predictor step is longer than this chosen value, then it is reduced to the upper bound. This maximal length can always be changed during the execution of the code as is explained below. If the user does not give any maximally allowed predictor length in the problem definition, then the upper bound is set to $+\infty$. The idea behind defining a maximal predictor length is to possibly improve the resolution of a path. Some steps may be bigger than the user may wish to have, therefore an upper bound is given here.

During the code execution, the most effective control feature of the implementation is the user prompt, which after `counter`-number of completed pathfollowing steps, will ask the user how to proceed. The user can then choose to either

1. enter `n`: then the user will be prompted to enter a natural number, which will change the default number of pathfollowing steps executed at a time, to this number. From now on, the number of pathfollowing steps being executed before the user is prompted for the next action, is this number chosen by the user, until the user may choose to change it again;
2. enter `p`: the user is prompted to enter the desired real number which bounds the length of the predictor step from this point onwards;
3. enter `f`: the user is prompted to enter a row-vector of steps, i.e. `[0 7 15 178]`, or simply a number, i.e. `23`, which refers to an already computed step. For each of the chosen steps, the computed approximation will be displayed in a new figure. Afterwards, the user can again choose any of the 5 options enumerated here;

4. enter `s`: this will save the run in the folder and with the name that were given in the problem definition file. Before halting the run, the code will display the name and location at which the file has been saved. Then, it will give a brief summary of the run, consisting of the value of λ^*, at which the run started, the values of λ^*, at the turning points, if some were encountered, the value of λ^* at which the run ended, the number of steps that were performed and finally whether mesh adaptation was used and how it was used;
5. or simply press `enter`: the code will compute the number of steps being executed at a time, which is either 1 or the number to which the user changed it, and then prompt the user again what to do next.

Another important feature, which can be turned on or off before the code is executed, is the possibility the code gives to move any number of steps backwards along the path. The way this feature works is that in each step, after the corrected point is computed on the path, some key values of the just computed step are saved, which allow to later come back to this step. The computation continues then as usual, for the chosen number of steps `counter`, and then after the plots of the newly computed results are displayed but just before prompting the user what to do next, the user is prompted to choose whether to continue or to move back a number of steps. This feature may be convenient, when a problem is computed for the first time. The automatically chosen step-length could turn up to be too optimistic, which may lead to undesired results or possibly strong mesh modification and thus large computational effort from this point forward. This may be overcome by going back a few steps, saving the results up to this point and then continue with adjusted solver settings along the path. If this feature is not needed, the variable `save_ws` in line `5` of the file `pathfollowing.m` can be set to `0`, which will suppress the user prompt to be displayed.

After each completion of the set number of pathfollowing steps, a number of plots are displayed, if these are enabled in the lines `6` to `11` of the file `pathfollowing.m`. These plots are

1. the plot of the evolution of the characteristic value of the solution, which is followed along the variation of the pathfollowing parameter, and the three dimensional solution evolution plot, along the variation of the pathfollowing parameter,
2. the evolution of the mesh in each step and the three dimensional evolution plot of the mesh density in each step over the problem interval,
3. the log plot, which contains the plots of all the values that are logged during each step of the pathfollowing. These plots show the evolution of

   - the CPU time to compute a single step and a marker for each step in which the trust-region method was called, and thus the step-length halved and the step recomputed,
   - the step-length which was used in each step, the predicted step-length, before the prediction-correction procedure, and a marker for each step in which the length of the predictor step was bounded by the value of `max_pred_length`,
   - θ_0 and a fixed line for $\theta_{\max}$, as both defined in (1.19),

- $\theta_{\max}/\theta_0$,
- the number of mesh points at the beginning and at the end of each step, and also a marker for each step in which the condition set in `'meshFactorMax'` was not satisfied, and thus the step-length halved and the step recomputed,
- the norms of $\boldsymbol{F}$ evaluated at the predictor point and at the corrected point,
- the absolute value of the last entry in the vectors $\boldsymbol{F}$ evaluated at the predictor point and at the corrected point,
- the norm of the first Newton increment, the norm of the second simplified Newton increment and the norm of the distance between the predictor and the corrected point, and also a marker for each step in which the condition set in the solver settings file in`'PredLengthFactor'` was not satisfied, a marker for each step in which the condition set in `'CorrLengthFactor'` was not satisfied, in which cases the step-length was halved and the step recomputed,
- the value c_s as defined in (1.12), the value of the cosine of the angle between a step and the tangent in the following step, and a marker for each step in which the condition set in `'anglemin'` was not satisfied, and thus the step-length halved and the step recomputed, and finally
- the maximal value of the error estimate for each corrected step.

Another important feature is to plot the solution at any specific value of the pathfollowing parameter during the run, as specified in `require_exact` in the problem definition file. Whenever one of these values is passed, the program will compute the solution for the specific pathfollowing parameter value and then afterwards continue from the last computed point.

Once a computation of a path is completed and saved, the user may want to restart from the last point in the path by simply changing in the problem definition file `startat` to the name of the file, which must be located in the folder which is given under `dir`. This file is then loaded and the pathfollowing resumes as usual from this state. This feature can be used to change some pathfollowing related parameters in the problem definition and in the solver settings. Even the function `PathCharData` can be set to follow a different characteristic value of the solution to the problem. The evolution of the newly set characteristic values is computed for the known path and from then on the values returned by this new function are saved. This feature is demonstrated for a pathfollowing problem in Section 2.2. Changing settings as for instance the number of collocation points which are used during the solution approximation, may lead to errors and is not recommended.
Furthermore, with an already computed path, by setting the option `only_exact` to `1`, the approximations to the solution of the BVP at the requested values for λ^*, saved in `require_exact`, are computed. Once this is done, the path, which was not altered, is saved and also the solution data at the requested values, in `p_save` and `p_exact` respectively. In other words, this feature allows to find approximations to solutions of the BVP along the already computed path, even for values that were not computed during the pathfollowing run.

During the development of the code, it was also discussed to reduce the number of mesh points during the pathfollowing procedure, when mesh adaptation is enabled and the current number of mesh points is not needed anymore to achieve satisfactory results. This feature is present in the code, in the lines `916-934` of the file `pathfollowing.m`, but was never used in a convincing manner. The reason for that was that the error tolerances for the solutions to problems were usually always harder to achieve the more pathfollowing steps would be computed. Thus a refinement of the mesh was observed over time, and making the mesh coarser again would not prove useful in the studied cases. Surprisingly the mesh adaptation would not happen, as could be expected, around turning points, but sometimes at other instances of the run. Although seemingly in some cases mesh adaptation is needed anyway, often controlling the step-length more cautiously seems to address the issue of the mesh getting too fine reasonably well for the tested examples. This feature is disabled but remains present in the code for possible future use for a fitting problem.

1.3 Test examples

In this section the examples on which the implementation was tested are presented. The problems that are presented here are:

The main aim of these test example was to reproduce previously published results with or newly implemented adaptive algorithm. The Complex Ginzburg-Landau equation is a parameter dependent problem and the hydrodynamics model is a problem posed on $[0, \infty)$. Thus, these examples also served to test the pathfollowing for different types of problems which `bvpsuite2.0` is equipped to handle.

The computation of these examples will mark the end of the presentation of the newly implemented pathfollowing module. In the next chapter, the changes in the code and also some tests addressing these changes are discussed.

Bratu equation

The first example is a relatively simple problem which was taken from [9]. Consider the BVP

$$\begin{aligned} &z''(t) + \lambda^* e^{z(t)} = 0 \qquad \text{for } t \in [0,1]\,, \\ &z(0) = 0, \quad z(1) = 0. \end{aligned}$$

Due to its simplicity it may serve as a first test example. The aim was to reproduce the path shown in Figure 1 from [9].
The absolute and relative tolerances of the Newton iteration are set to 10^{-6}, the absolute and relative tolerances of the mesh adaptation to 10^{-4}. The computations are started with a mesh of 51 equidistant points on the interval $[0,1]$ and 3 Gaussian collocation points are used to approximate the problem's solution. The characteristic value of the solution which is followed along the path is $z'(0)$. This is achieved by the function

```
function ret = PathCharData(x1,coeff,ordnung,rho)
ret = coeffToValues( coeff,x1,ordnung,rho,0,1);
end
```

In order to illustrate how $\theta_{\max}$ influences the pathfollowing behaviour, this example is computed for the values $\theta_{\max} = 10^{-2}$ and $\theta_{\max} = 10^{-3}$. The aim was to follow the characteristic solution value up to $z'(0) = 50$, which was set with the optional settings `pit_stop` in the problem definition file. The starting value for λ^* is 0 and the starting step-length 1, meaning that λ^* will grow in positive direction. The target was reached after 72 steps for the run with $\theta_{\max} = 10^{-2}$ and after 187 steps for the run with $\theta_{\max} = 10^{-3}$. The results are displayed in Figure 1.3 and Figure 1.4, respectively.
In both cases, mesh adaptation is enabled and the mesh was adapted at approximately the same value of the pathfollowing parameter along the path. In the run with $\theta_{\max} = 10^{-2}$, the mesh adaptation took place at step 37 and $\lambda^* \approx 5.97 \times 10^{-3}$. The mesh was adapted to 97 points. In the run with $\theta_{\max} = 10^{-3}$, the mesh adaptation took place at step 82 and $\lambda^* \approx 8.11 \times 10^{-3}$. The mesh was adapted to 95 points. Points are getting added to the mesh, but they remain equidistributed over the interval, as is shown in Figure 1.5.
The smaller $\theta_{\max}$ is chosen, the finer the resolution gets. Along the run in both cases, an adaptation of the step-length around the turning point and beyond is visible. The mesh adaptation interestingly takes place for approximately the same value and the smaller step sizes from the run with $\theta_{\max} = 10^{-3}$ do not seem to have any effect on it.

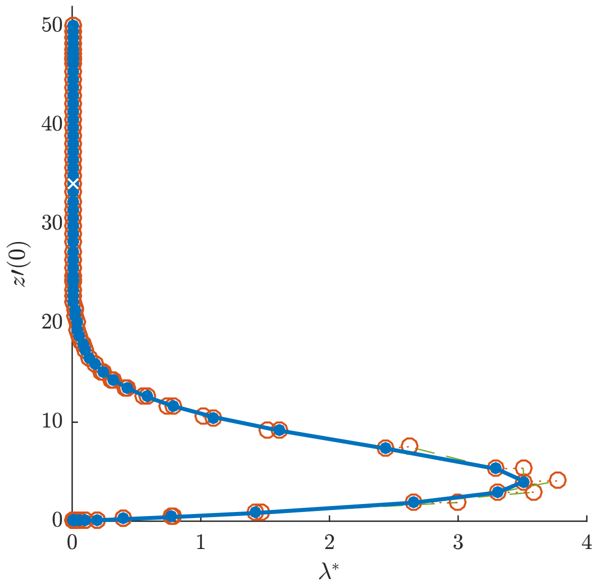


Figure 1.3: Bratu equation: Evolution of $z'(0)$ under variation of λ^* with $\theta_{\max} = 10^{-2}$, until $z'(0) = 50$.

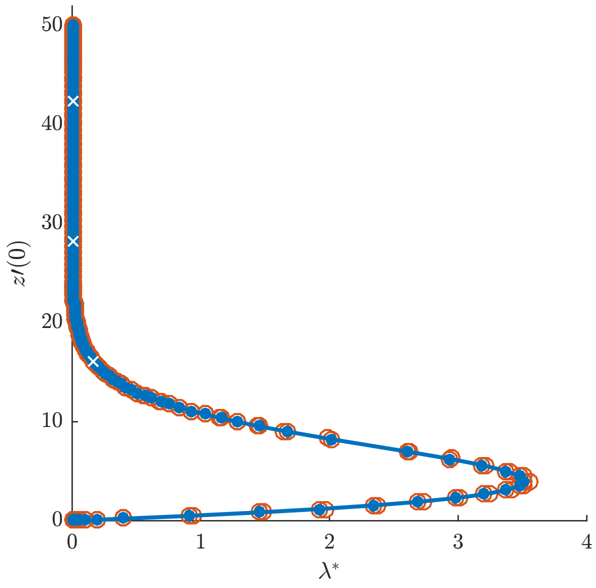


Figure 1.4: Bratu equation: Evolution of $z'(0)$ under variation of λ^* with $\theta_{\max} = 10^{-3}$, until $z'(0) = 50$.

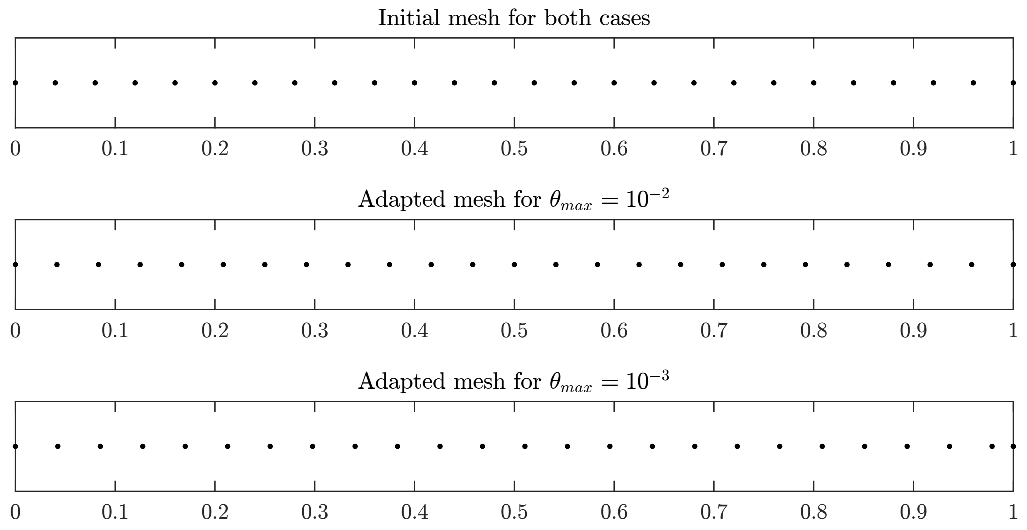


Figure 1.5: Bratu equation: Distribution of the mesh points over the interval in both cases before and after mesh adaptation.

Equation of a catalytic reaction in a flat particle

This example from [4] is quite similar to the first example, the Bratu equation. Consider the BVP

$$\begin{aligned} z''(t) &= \lambda^* \exp\left(\frac{8(1-z(t))}{1+0.4(1-z(t))}\right) \qquad \text{for } t \in [0,1]\,, \\ z'(0) &= 0, \quad z(1) = 1. \end{aligned}$$

The aim was to reproduce the path shown in Figure 3 from [4].
As in the first example, the absolute and relative tolerances of the Newton iteration are set to 10^{-6}, the absolute and relative tolerances of the mesh adaptation to 10^{-4}. The computations are started with a mesh of 51 equidistant points on the interval $[0,1]$ and 3 Gaussian collocation points are used to approximate the problem's solution. The characteristic value of the solution which is followed along the path is $z(0)$. This is achieved by defining in the problem definition the local function

```
function ret = PathCharData(x1,coeff,ordnung,rho)
ret = coeffToValues( coeff,x1,ordnung,rho,0,0);
end
```

The starting value for λ^* is 0 and the starting step-length 10^{-2}. $\theta_{\max}$ is set to 10^{-2}. In order to reproduce the results from Figure 3 from [4], the aim was to follow the pathfollowing parameter up to the value $\lambda^* = 0.3$, which was set with the optional settings `pit_stop` in the problem definition file. The value was reached after 26 steps, without any mesh adaptation. The resulting path is displayed in Figure 1.6.
This example is quite similar to the first one. It having two turning points was another test which the code dealt well with.

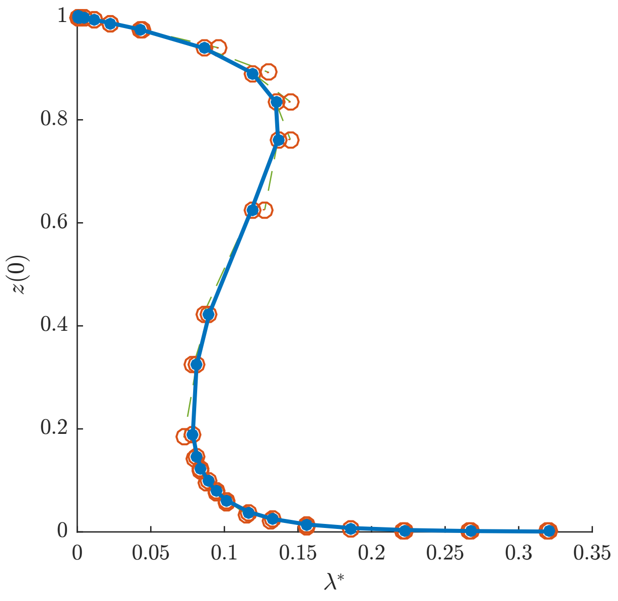


Figure 1.6: Catalytic reaction: Evolution of $z(0)$ under variation of λ^* with $\theta_{\max} = 10^{-2}$, until $\lambda^* = 0.3$.

Shell buckling problem

This example was computed in [4], the results from this computation, which are aimed to be reproduced, can be found in Figure 5 from [4].
It is a singular BVP and also a more challenging problem than the previous ones. Consider the BVP

$$\begin{aligned}
&\delta\Bigg(z_1''(t)+z_1'(t)\cot(t)+\cot(t)^2\frac{\cos(t-z_1(t))}{\cos(t)}\frac{\sin(t-z_1(t))-\sin(t)}{\cos(t)}\\
&\qquad-0.3\frac{\cos(t-z_1(t))-\cos(t)}{\sin(t)}\Bigg)\\
&\quad=-z_2(t)\frac{\sin(t-z_1(t))}{\sin(t)}-4\lambda^*\frac{\cos(t-z_1(t))}{\sin(t)}z_3(t), \qquad (1.43a)\\
&\delta\left(z_2''(t)+z_2'(t)\cot(t)-z_2(t)\left(\cot(t)^2\frac{\cos(t-z_1(t))^2}{\cos(t)^2}-0.3(1-z_1'(t))\frac{\sin(t-z_1(t))}{\sin(t)}\right)\right)\\
&\quad=\frac{\cos(t-z_1(t))-\cos(t)}{\sin(t)}+\delta\Bigg(-4\lambda^*\cot(t)z_3(t)\\
&\qquad\cdot\left(\frac{\sin(2(t-z_1(t)))}{\sin(2t)}+0.3(1-z_1'(t))\frac{\cos(t-z_1(t))}{\cos(t)}\right)\\
&\qquad+4\lambda^*\frac{(\sin(t)^2(1-z_1'(t))\cos(t-z_1(t))+2\sin(t)\cos(t)\sin(t-z_1(t))}{\sin(t)}\Bigg), \qquad (1.43b)\\
&z_3'(t)=\cos(t-z_1(t))\sin(t), \qquad \text{for } t\in[0,\pi] \qquad (1.43c)\\
&z_1(0)=z_1(\pi)=0, \quad z_2(0)=z_2(\pi)=0, \quad z_3(0)=0, \qquad (1.43d)
\end{aligned}$$

where $\delta = 0.00369$.
This problem has two very close solution paths around the parameter value $\lambda^* = 1$, i.e. a bifurcation point. Close to $\lambda^* = 1$, whenever the step is too big, one path is left and the other followed from then on. The automatic adaptive step-length control takes care of staying on one path throughout, when the right solver settings are chosen.
Here, the absolute and relative tolerances of the Newton iteration are set to 10^{-6}, the absolute and relative tolerances of the mesh adaptation to 10^{-4}. The computations are started with a mesh of 51 equidistant points on the interval $[0, \pi]$ and 3 Gaussian collocation points are used to approximate the problem's solution. The characteristic value of the solution which is followed along the path is $\|z_1\|_\infty$. This is achieved by defining in the problem definition file the local function

```
function ret = PathCharData(x1,coeff,ordnung,rho)
help = coeffToValues( coeff, x1,ordnung,rho,x1,0);
ret = max(help(1,:));
end
```

The starting value for λ^* is 0 and the starting step-length 1. Also, $\theta_{\max}$ is set to 5×10^{-2}, `'maxSteplengthGrowth'` to 4, `'meshFactorMax'` to 2, `'PredLengthFactor'` and `'CorrLengthGrowth'` to 4. The other settings are kept to their defaults. The results are displayed in Figure 1.7. Here some mesh adaptation has taken place.

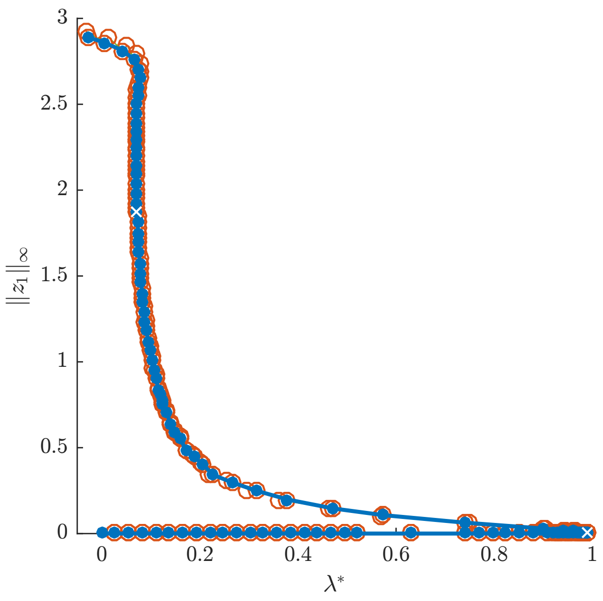


Figure 1.7: Shell buckling problem: Evolution of $\|z_1\|_\infty$ under variation of λ^* with $\theta_{\max} = 5 \times 10^{-2}$, until $\lambda^* = 0$. This is the path of the double dimple solution

As mentioned above, the difficulty of this example lies in the turning point near 1. Until the values are quite close to 1, $\|z_1\|_\infty$ remains on the order of magnitude on the scale of 10^{-10}. At $\lambda^* = 0.95$, $\|z_1\|_\infty$ is still about $3 * 10^{-9}$ and even for $\lambda^* = 0.98$ it is about $5 * 10^{-7}$. The turning point is reached at about $\lambda^* = 0.99$ and $\|z_1\|_\infty = 4 * 10^{-5}$. After reaching the turning point, the step-lengths increases again, but there is still some mesh adaptation required to control the error distribution of the approximation, in accordance with the required absolute and relative tolerances of 10^{-4}.
In this example the implementation on the restrictions on the length of the corrector step compared to the length of the corrector step in the previous step was of crucial importance for the autonomous progress along the path. This restriction withheld the algorithm from taking leaps that would cause a change of paths. In this example, until the turning point was reached, the length of the predictor steps was always much larger than the length of the corrector step, since $\|z_1\|_\infty$ was almost zero until then. So this restriction, on the other hand, did not help in this example.
All in all, the adjustments in the solver settings made the adaptive step-length control strategy compute this problem, without any intervention of the user after providing the problem definition and solver settings. Therefore the implemented features may also be valuable for future challenging examples.
Note, that the results from Figure 5 from [4] and from Figure 1.7 are not similar beyond $\|z_1\|_\infty = 2.5$ approximately. The reason is that in [4] the so-called single dimple solution

was found, and here the double dimple solution was found. By trying different solver settings, this seems to be in strong correlation with the chosen tolerances, which would lead in our computations to find the path of the double dimple solution. By considering an eigenvalue problem, which is found from (1.43), some starting profiles are found. Through [19], the starting profiles displayed in Figure 1.8 were obtained. These then led to the path of the single dimple solution, displayed in Figure 1.9.

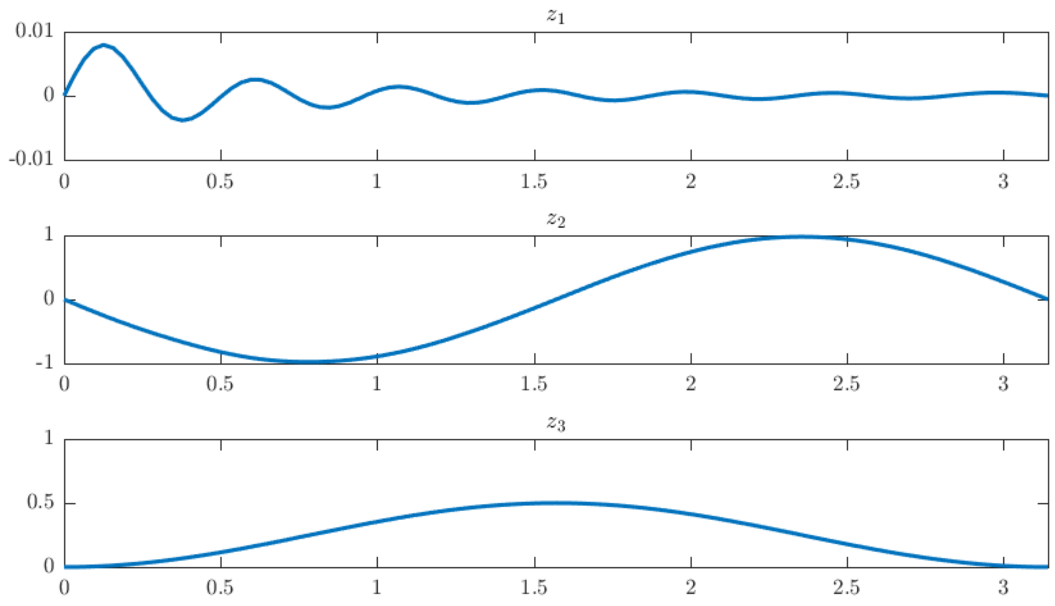


Figure 1.8: Shell buckling problem: Starting profiles for z_1, z_2 and z_3 for the single dimple solution.

For the two paths in Figure 1.7 and Figure 1.9, the shape of the resulting ball was computed through the BVP

$$x'(t) = \cos(t - z_1(t)), \quad y'(t) = \sin(t - z_1(t)), \qquad \text{for } t \in [0, \pi], \tag{1.44a}$$

$$x(0) = 0, \quad y\left(\frac{\pi}{2}\right) = 0. \tag{1.44b}$$

The solutions x and y are then the coordinates of the right half of the ball. Computing these coordinates for each step on the path, i.e. inserting z_1 from each step into the BVP (1.44), animations were created showing the deformation of the ball in case of the double dimple solution, in Figure 1.10, and in case of the single dimple solution, in Figure 1.11.

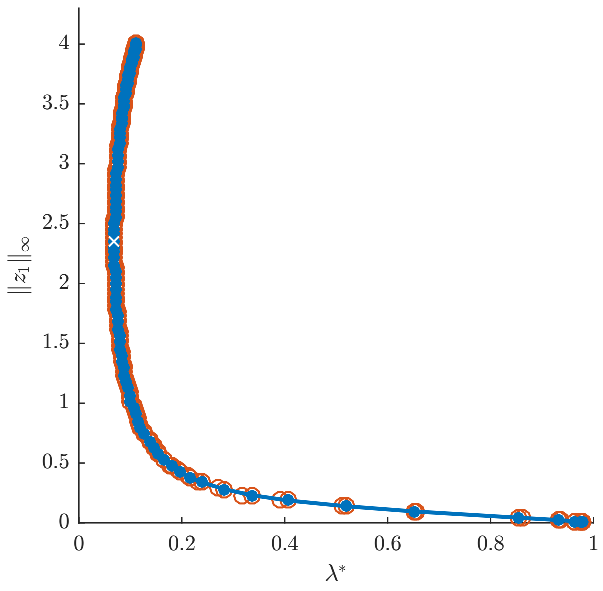


Figure 1.9: Shell buckling problem: Evolution of $\|z_1\|_\infty$ under variation of λ^* with $\theta_{\max} = 5 \times 10^{-2}$, until $\|z_1\|_\infty = 4$. This is the path of the single dimple solution.

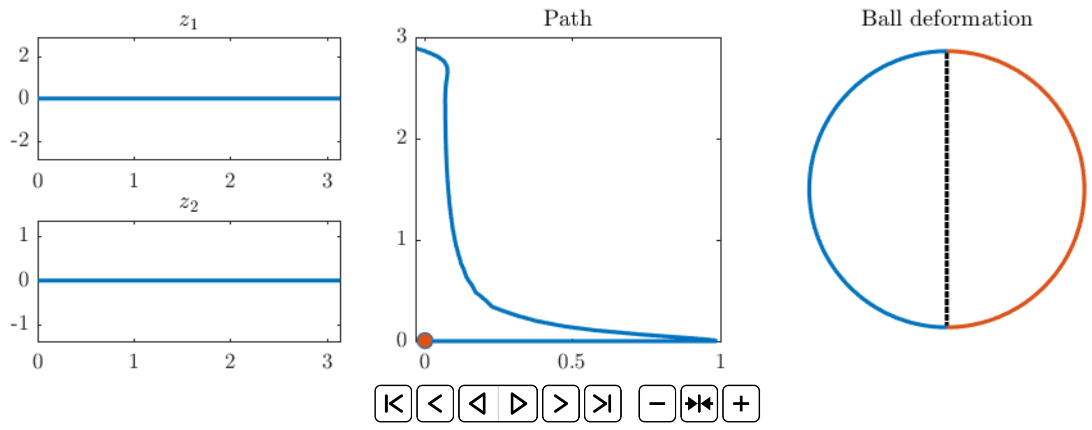


Figure 1.10: Shell buckling problem: Animation of the ball along the path of the double dimple solution.

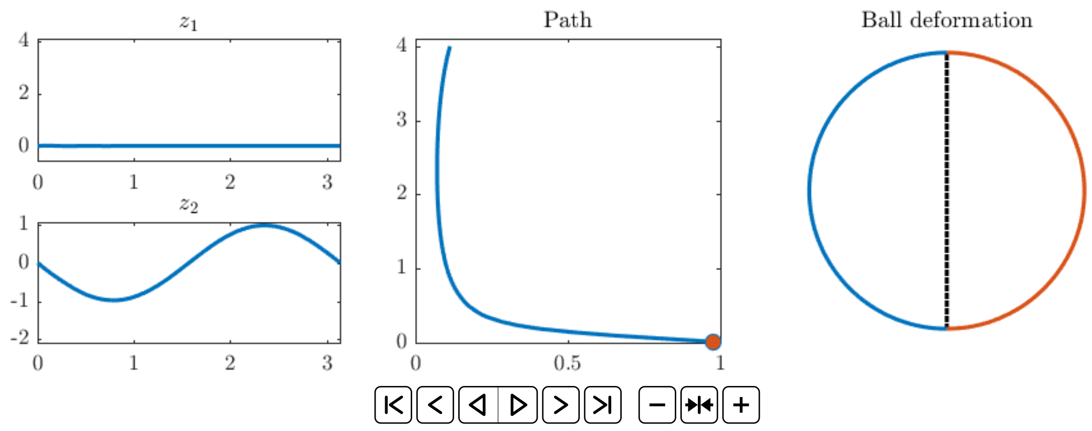


Figure 1.11: Shell buckling problem: Animation of the ball along the path of the single dimple solution.

Complex Ginzburg-Landau equation

This, also taken from [4], is the complex Ginzburg-Landau equation. The non-linear ODE for $t > 0$ and the boundary conditions are

$$(1 - i\lambda_1^*)\left(z''(t) + \frac{2}{t}z'(t)\right) - z(t) + ip(tz'(t) + z(t)) + (1 + i\lambda_2^*)|z(t)|^2 z(t) = 0, \quad (1.45a)$$

$$z'(0) = 0, \quad \Im(z(0)) = 0, \quad \lim_{t\to\infty} tz'(t) = 0. \quad (1.45b)$$

The solution function $z(\cdot)$ is complex-valued. In addition to the function $z(\cdot)$, the unknown parameter p needs to be determined from this system. The system contains two pathfollowing parameters, λ_1^* and λ_2^*, which can be varied independently. The notation $|x + iy| := \sqrt{x^2 + y^2}$ for $x, y \in \mathbb{R}$ denotes the modulus of a complex number, also called the complex norm. In order to rewrite this system in terms of real functions, denote

$$z(t) =: z_1(t) + iz_2(t) \text{ with } z_1, z_2 : \big[0, \infty)\big) \to \mathbb{R}.$$

Then the equation above is written as the mixed order system

$$\begin{aligned} &tz_1''(t) + 2z_1'(t) + \lambda_1^*(tz_2''(t) + 2z_2'(t)) - tz_1(t) - tp(tz_2'(t) + z_2(t)) \\ &\qquad + t(z_1(t)^2 + z_2(t)^2)(z_1(t) - \lambda_2^* z_2(t)) = 0, && (1.46a) \\ &tz_2''(t) + 2z_2'(t) - \lambda_1^*(tz_1''(t) + 2z_1'(t)) - tz_2(t) + tp(tz_1'(t) + z_1(t)) \\ &\qquad + t(z_1(t)^2 + z_2(t)^2)(\lambda_2^* z_1(t) + z_2(t)) = 0, \qquad \text{for } t \in [0, \infty), && (1.46b) \end{aligned}$$

with the boundary conditions

$$z_1'(0) = 0, \quad z_2'(0) = 0, \quad z_2(0) = 0, \quad (1.47)$$

$$\lim_{t\to\infty} tz_1'(t) = 0, \quad \text{and} \quad \lim_{t\to\infty} tz_2'(t) = 0. \quad (1.48)$$

Due to the boundary conditions set for $t \to \infty$, in order to be able to compute an approximate solution with `bvpsuite2.0`, the problem is redefined on the interval $[0, 1]$. This is done by keeping the system (1.46) for $t \in [0, 1]$ and for $t \in [1, \infty)$ defining

$$\tau := \frac{1}{t}.$$

Then, τ assumes values in $(0, 1]$. For a function $\tilde{z} : [0, \infty) \to \mathbb{R}$ the equations

$$\begin{aligned} &\frac{\mathrm{d}\tau}{\mathrm{d}t} = -\frac{1}{\tau^2}, \quad \tilde{z}(t) = \tilde{z}(\tfrac{1}{\tau}), \\ &\frac{\mathrm{d}\tilde{z}(\frac{1}{\tau})}{\mathrm{d}\tau} = -\frac{\mathrm{d}\tilde{z}(t)}{\mathrm{d}t}\frac{1}{\tau^2} \quad \Rightarrow \quad \frac{\mathrm{d}\tilde{z}(t)}{\mathrm{d}t} = -\tau^2\frac{\mathrm{d}\tilde{z}(\frac{1}{\tau})}{\mathrm{d}\tau}, \\ &\frac{\mathrm{d}^2\tilde{z}(\frac{1}{\tau})}{\mathrm{d}\tau^2} = \frac{\mathrm{d}}{\mathrm{d}\tau}\left(-\frac{\mathrm{d}\tilde{z}(t)}{\mathrm{d}t}\frac{1}{\tau^2}\right) = \frac{1}{\tau^4}\frac{\mathrm{d}^2\tilde{z}(t)}{\mathrm{d}t^2} - \frac{2}{\tau}\frac{\mathrm{d}\tilde{z}(\frac{1}{\tau})}{\mathrm{d}\tau} \\ &\qquad \Rightarrow \quad \frac{\mathrm{d}^2\tilde{z}(t)}{\mathrm{d}t^2} = \tau^4\frac{\mathrm{d}^2\tilde{z}(\frac{1}{\tau})}{\mathrm{d}\tau^2} + 2\tau^3\frac{\mathrm{d}\tilde{z}(\frac{1}{\tau})}{\mathrm{d}\tau}. \end{aligned}$$

hold. Thus, for $\tau \in (0,1]$, (1.46) can be written as

$$\begin{aligned}\tau^4 z_1''(\tfrac{1}{\tau}) + \lambda_1^* \tau^4 z_2''(\tfrac{1}{\tau}) - z_1(\tfrac{1}{\tau}) + p(\tau z_2'(\tfrac{1}{\tau}) - z_2(\tfrac{1}{\tau}))& \\ + \tau(z_1(\tfrac{1}{\tau})^2 + z_2(\tfrac{1}{\tau})^2)(z_1(\tfrac{1}{\tau}) - \lambda_2^* z_2(\tfrac{1}{\tau})) = 0,& \qquad (1.49a)\\ \tau^4 z_2''(\tfrac{1}{\tau}) - \lambda_1^* \tau^4 z_1''(\tfrac{1}{\tau}) - z_2(\tfrac{1}{\tau}) - p(\tau z_1'(\tfrac{1}{\tau}) - z_1(\tfrac{1}{\tau}))& \\ + \tau(z_1(\tfrac{1}{\tau})^2 + z_2(\tfrac{1}{\tau})^2)(\lambda_2^* z_1(\tfrac{1}{\tau}) + z_2(\tfrac{1}{\tau})) = 0.& \qquad (1.49b)\end{aligned}$$

Now, define $z_3(\tau) := z_1(\frac{1}{\tau})$ and $z_4(\tau) := z_2(\frac{1}{\tau})$ and the continuity conditions

$$z_3(1) = z_1(1), \quad z_3'(1) = -z_1'(1), \quad z_4(1) = z_2(1) \quad \text{and} \quad z_4'(1) = -z_2'(1), \qquad (1.50)$$

are imposed.

The boundary conditions (1.48) in the limit to infinity cannot be properly expressed in `bvpsuite2.0`, therefore the functions $z_5(\tau)$ and $z_6(\tau)$ governed by the equations

$$z_5(\tau) = \tau z_3'(\tau), \quad \text{and} \quad z_6(\tau) = \tau z_4'(\tau) \quad \text{for } \tau \in [0,1] \qquad (1.51)$$

are added to the system, with the boundary conditions

$$z_5(0) = 0 \quad \text{and} \quad z_6(0) = 0. \qquad (1.52)$$

With z_5, z_6 and the relations

$$z_3''(\tau) = \frac{1}{\tau}\left(z_5'(\tau) - z_3'(\tau)\right) \quad \text{and} \quad z_4''(\tau) = \frac{1}{\tau}\left(z_6'(\tau) - z_4'(\tau)\right),$$

(1.49) can be rewritten as

$$\begin{aligned}\tau^3 z_5'(\tau) - \tau^2 z_5(\tau) + \lambda_1^*\left(\tau^3 z_6'(\tau) - \tau^2 z_6(\tau)\right) - z_3(\tau) + p(\tau z_4'(\tau) - z_4(\tau))& \\ + (z_3(\tau)^2 + z_4(\tau)^2)(z_3(\tau) - \lambda_2^* z_4(\tau)) = 0,& \qquad (1.53a)\\ \tau^3 z_6'(\tau) - \tau^2 z_6(\tau) - \lambda_1^*\left(\tau^3 z_5'(\tau) - \tau^2 z_5(\tau)\right) - z_4(\tau) - p(\tau z_3'(\tau) - z_3(\tau))& \\ + (z_3(\tau)^2 + z_4(\tau)^2)(\lambda_2^* z_3(\tau) + z_4(\tau)) = 0.& \qquad (1.53b)\end{aligned}$$

Combining all of these considerations, the six equations from (1.46) for $t \in [0,1]$ and (1.51) and (1.53) for $\tau \in (0,1]$ with the nine boundary conditions from (1.47), (1.50) and (1.52) are equivalent to the BVP (1.45). Furthermore, in this form solutions can be computed by `bvpsuite2.0`.

When $\lambda_1^* = 0$ and $\lambda_2^* = 0$, then (1.45) is the nonlinear Schrödinger equation and there exist more than one solution. The aim in this example was originally to reproduce a part of the work in Georg Kitzhofers Doctoral thesis [3], where $\lambda_2^* = 0$ is fixed and λ_1^* is varied, but some obstacles had to be avoided. In the thesis, Kitzhofer starts the pathfollowing procedure at the unstable bumpy solutions $(0,1)$ and $(1,3)$ – written in the notation introduced in [12]. The initial solution $(0,1)$ was reported to start from relatively simple initial solutions. Trying this approach was not successful. The initial

solution $(1,3)$ was found using a shooting method. This was not implemented in the present work. A solution which was found with `bvpsuite2.0` and the equations above is the $(1,1)$ solution, which is stable and monotone, meaning $|z(t)| \to 0$ as $t \to \infty$. This is the solution on which the focus lies in the following pages.
The aim was thus shifted to reproducing a similar behaviour for the pathfollowing procedure starting at the $(1,1)$-solution as in the graph presented in Figure 3.6 from [13], where the paths of other solutions of the equation (1.45) are also displayed. $\lambda_2^* = 0$ is fixed and λ_1^* is varied.
The characteristic value of the solution that is followed is the evolution of the parameter p. This is achieved by defining in the problem definition file the local function

```
function ret = PathCharData(~,coeff,~,~)
ret = coeff(end-1);
end
```

The starting value for λ_1^* is 0 and the starting step-length 1. Here, the absolute and relative tolerances of the Newton iteration are set to 10^{-8}. Mesh adaptation is not activated for this example. The computations start with a mesh with 281 points, which are placed at a distance of 10^{-4} between 0 and 0.001, 10^{-3} between 0.001 and 0.01 and 10^{-2} between 0.01 and 1. 2 Gaussian collocation points are used to approximate the problem's solution. $\theta_{\max}$ was set to 0.1. The absolute value of the initial solution of the run is displayed in Figure 1.12, which corresponds to figure 5.5 in [3, p. 85]. The error estimation is turned on, in order to keep track of the error's evolution. The other values are all kept to their defaults. The resulting path is computed in 72 steps and is displayed in Figure 1.13.

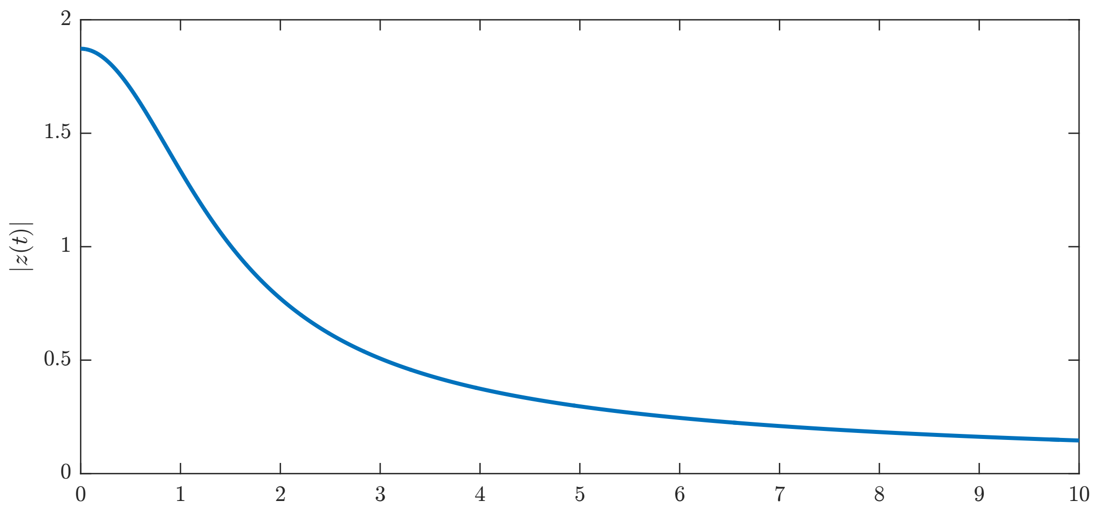


Figure 1.12: Complex Ginzburg-Landau equation: Absolute value of the initial solution $(1,1)$.

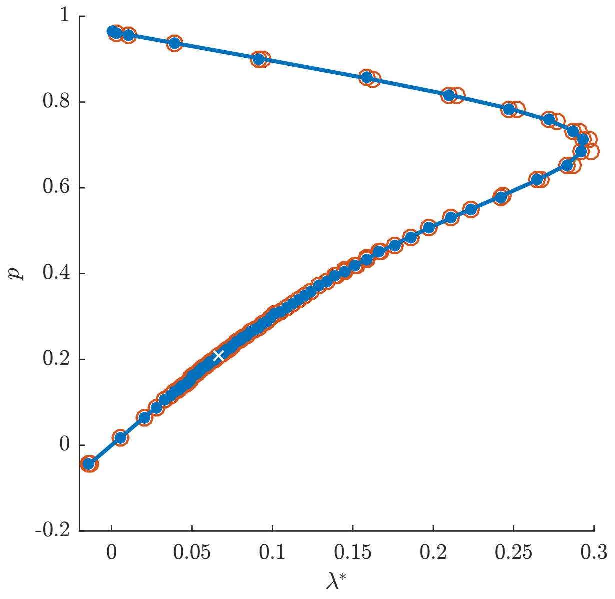


Figure 1.13: Complex Ginzburg-Landau equation: Evolution of the parameter p.

The pathfollowing parameter starts at the value $\lambda^* = 0$, where the parameter value is approximately $p \approx 9.63795893 \times 10^{-1}$. This value is not entirely the same as in the table in [3, p. 102], it differs by approximately 0.05. The value of λ^* is varied, and the value of p goes to 0. In $\lambda^* = p = 0$, the solution to the equation has imaginary part 0. The solution is then real valued. This can be seen in Figure 1.14.

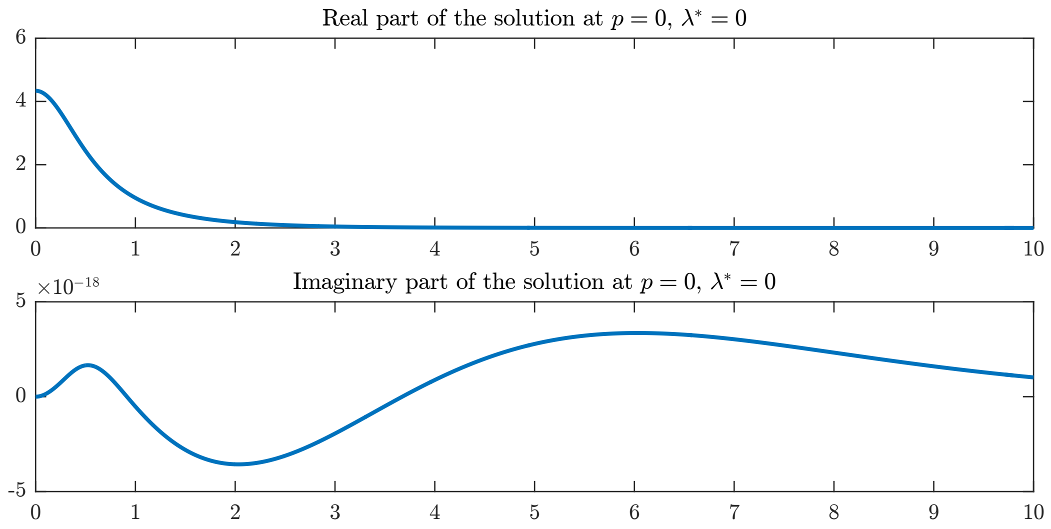


Figure 1.14: Complex Ginzburg-Landau equation: Solution for $p = 0$, $\lambda^* = 0$.

In this example, the main difficulty was the search for a good initial solution from where the path could start. This difficulty was not overcome within the scope of this work.

Hydrodynamics model

The pathfollowing problem reported in [8], is

$$\begin{aligned} &z''(t) + \frac{2}{t}z'(t) = 4(z(t)+1)z(t)(z(t)-\lambda^*), \\ &z'(0) = 0, \quad z(\infty) = \lambda^*. \end{aligned}$$

The boundary condition at infinity is a Dirichlet boundary conditions, thus it does not pose any problem to `bvpsuite2.0`. The aim in this example was to reproduce the solution profiles as shown in Figure 1 from [8].
The absolute and relative tolerances of the Newton iteration are set to 10^{-6}, the absolute and relative tolerances of the mesh adaptation to 10^{-4}. The computations are started with a mesh of 101 equidistant points on the interval $[0, 1]$ and 3 Gaussian collocation points are used to approximate the problem's solution. The characteristic value of the solution which is followed along the path is $z(0)$. The initial solution that was found was the solution for the parameter value $\lambda^* = 0.5$. Therefore, the starting step-length was once set to -0.01 and in a second run to 0.01, meaning that the pathfollowing was once computed in the direction towards 0 and once in the direction towards 1. Towards 0 the pathfollowing parameter was followed up to the value $\lambda^* = 0.005$ and towards 1 up to the value $\lambda^* = 0.9$, which was both set with the optional setting `pit_stop` in the problem definition file. In both cases, $\theta_{\max}$ was set to 10^{-2}. The value towards 0 was reached after 40 steps, without any mesh adaptation, and towards 1 after 125 steps, where the mesh was adapted as displayed in Figure 1.15. The results are displayed in Figure 1.16 and in Figure 1.17, respectively.

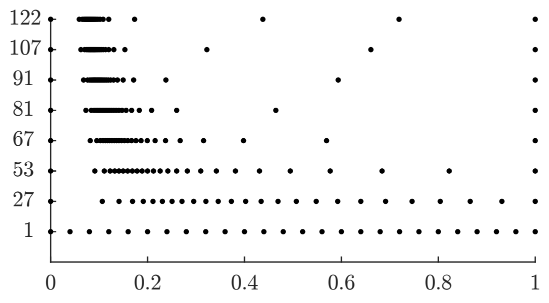


Figure 1.15: Hydrodynamics model: Distribution of the mesh points along the path from start (at the bottom) to finish (at the top).

The profiles from Figure 1.17 were computed with the help of the field `require_exact` set to `[0.005 0.1:0.1:0.9]`. Then the profiles could be easily accessed through the saved variable `p_exact`.
This test example posed on $[0, \infty)$ was helpful to rectify the code where it was needed for problems posed on semi-infinite intervals requiring mesh adaptation. The mesh adaptation also exhibited a special behaviour for this text example.

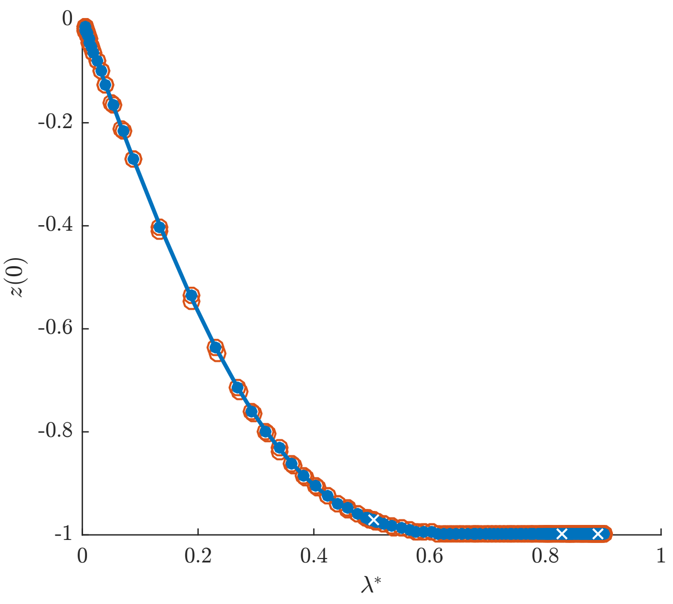


Figure 1.16: Hydrodynamics model: Evolution of $z(0)$ under variation of λ^*.

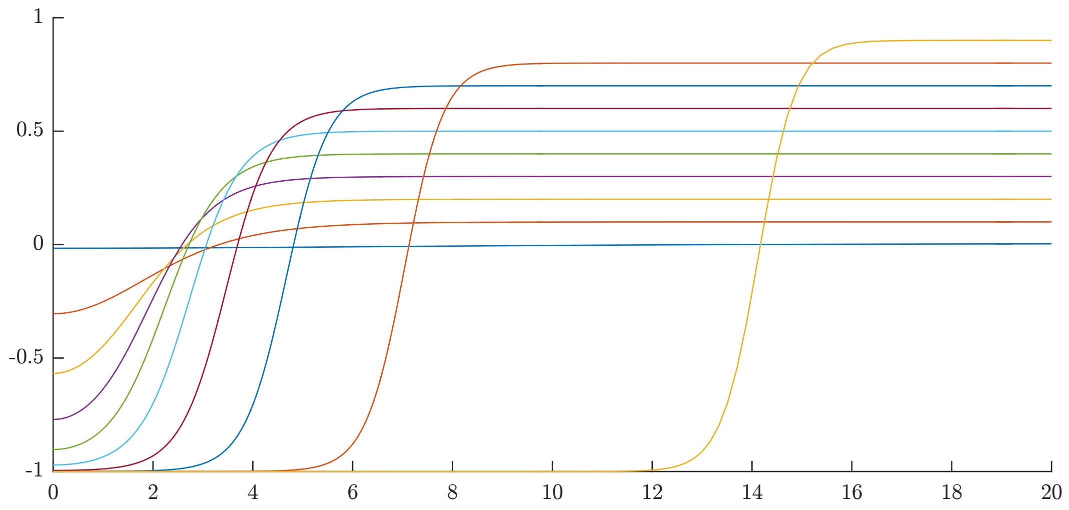


Figure 1.17: Hydrodynamics model: Solution profiles for $\lambda^* = 0.005, 0.1, ..., 0.9$.

Forced non-linear oscillator model

From [9], consider the forced non-linear oscillator modelled by the BVP

$$\left(\frac{\lambda^*}{2\pi}\right)^2 z''(t) + \frac{1}{25}\left(\frac{\lambda^*}{2\pi}\right) z'(t) - \frac{1}{5}z(t) + \frac{8}{15}z(t)^3 = \frac{2}{5}\cos(2\pi t) \quad \text{for} \quad t \in [0,1]\,,$$
$$z(0) = z(1), \quad z'(0) = z'(1).$$

The aim was here to reproduce Figure 2 from [9, p. 85]. This plot is quite entangled, which makes the pathfollowing difficult for this example.
The absolute and relative tolerances of the Newton iteration are set to 10^{-6}, the absolute and relative tolerances of the mesh adaptation to 10^{-4}. The computations are started with a mesh of 51 equidistant points on the interval $[0,1]$ and 2 Gaussian collocation points are used to approximate the problem's solution. The characteristic value of the solution which is followed along the path is $z(0)$. For the pathfollowing specific settings, $\theta_{\max}$ was set to 5×10^{-2}, `'maxCorrSteps'` to 10, in order to allow more corrections of the prediction for the next step-length, and `'PredLengthFactor'` to 1, which allowed the corrector step to be of the same length as the predictor step in the current step. During the pathfollowing, the solution would exhibit more and more oscillatory behaviour. This will eventually lead to mesh adaptation, where the mesh ended up with 499 points. Altogether, 410 pathfollowing steps were executed. The results are displayed in Figure 1.18.

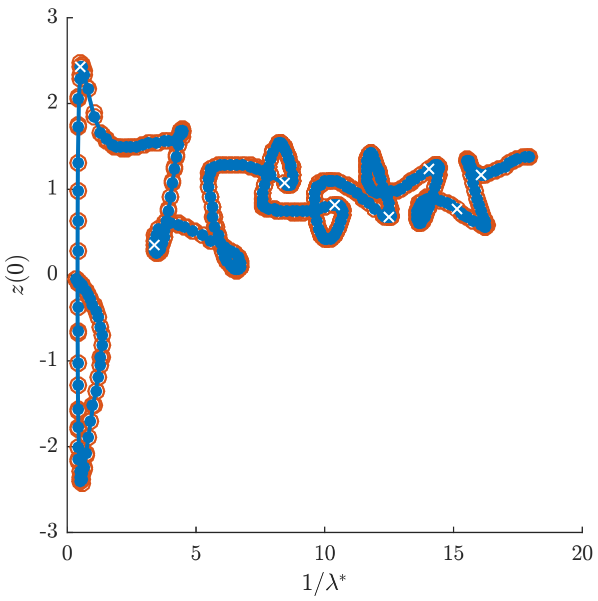


Figure 1.18: Forced non-linear oscillator: Evolution of $z(0)$ under variation of λ^*.

In this example, many bifurcation points are encountered. Interestingly enough, this did not seem to cause too much problems to the code. On the other hand, around the turning points, it was important for the step-length to not get too small, where seemingly it sufficed to relax the condition in `'PredLengthFactor'` to 1. Apparently, the bifurcation points along the path in this example seem to exhibit a behaviour that makes the pathfollowing code of `bvpsuite2.0` able to move along the path nicely.

Chapter 2

bvpsuite2.0

The aim of this chapter is to present the current state of the code at the release of the new package `bvpsuite2.0`.
In the first section, all the functions, which are included in the package, and their functionalities shall be detailed. In the second section, the functionality of each of the seven modules from `bvpsuite2.0` will be demonstrated through an example. These example runs can also be found in the manual explaining the use and functionality of `bvpsuite2.0` [16]. The manual was created over the course of the year by Merlin Fallahpour and Aron Sass, under supervision of Dr. Othmar Koch and Prof. Ewa B. Weinmüller.
The essential steps of the approximation of a solution to a BVP by `bvpsuite2.0` is shown in the flowchart diagram in Figure 2.1.

2.1 Functions of `bvpsuite2.0`

The aim of this section is to present the current code of `bvpsuite2.0` along with the changes it underwent during the implementation of the pathfollowing module. A brief summary of the special features of all functions from the `bvpsuite2.0` package, including pseudo-code of the lengthy functions, are provided in this section. This summary is meant for users of the code wanting to get a quick overview of all the functions within the code and also for developers to facilitate further improvements.
The functions presented in this section were developed in their original form at the Institute of Analysis and Scientific Computing of the University of Technology in Vienna by Dr. Georg Kitzhofer. His work named `bvpsuite1.1` built on the previously released collocation code `sbvp`, which was jointly developed by Prof. Winfried Auzinger, Dr. Othmar Koch and Prof. Ewa B. Weinmüller. `sbvp` was meant to handle singular problems, whereas `bvpsuite1.1` was meant for boundary value problems in a more general sense. Later on, Stefan Wurm and Markus Schöbinger rewrote the code `bvpsuite1.1` in its current modular form to improve readability of the code and its usability. Finally, in this work, the module for pathfollowing was implemented with an automatic step-length control strategy in order to round up and release the new version `bvpsuite2.0`.

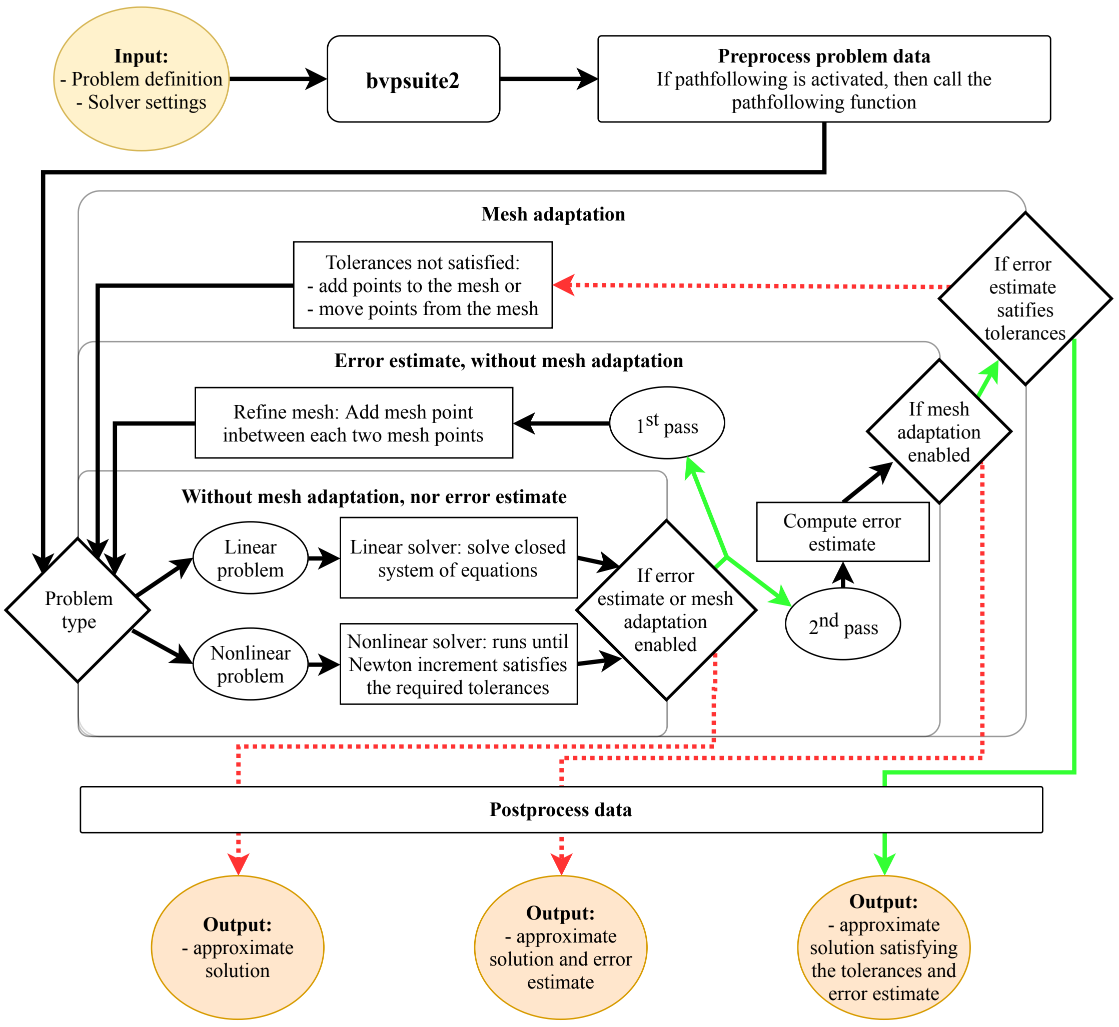

Input:
- Problem definition
- Solver settings
bvpsuite2
Preprocess problem data
If pathfollowing is activated, then call the pathfollowing function
Mesh adaptation
Tolerances not satisfied:
- add points to the mesh or
- move points from the mesh
If error estimate satifies tolerances
Error estimate, without mesh adaptation
Refine mesh: Add mesh point inbetween each two mesh points
1st pass
If mesh adaptation enabled
Without mesh adaptation, nor error estimate
Compute error estimate
Linear problem
Linear solver: solve closed system of equations
Problem type
If error estimate or mesh adaptation enabled
2nd pass
Nonlinear problem
Nonlinear solver: runs until Newton increment satisfies the required tolerances
Postprocess data
Output:
- approximate solution
Output:
- approximate solution and error estimate
Output:
- approximate solution satisfying the tolerances and error estimate

Figure 2.1: Flowchart of `bvpsuite2.0`.

The functions of `bvpsuite2.0` are enumerated chronologically in the order in which these are called when computing an approximation of the solution to a given problem. The following functions are provided in the package:

This list comprises all 19 functions provided in the `bvpsuite2.0` package, the lengthier ones with pseudo code. The description of the workings and the pseudo code of the two functions implemented for the pathfollowing module, `pathfollowing.m` and `functionFDF.m`, which were discussed exhaustively in the previous chapter, are not reiterated here.

`bvpsuite2`

The function `bvpsuite2` is the main function of `bvpsuite2.0`. It is called by the user and returns the approximation of the solution function. The mandatory arguments passed to the function are the problem definition and the solver settings, for which the templates `template_bvp` and `default_settings` respectively, are provided. These two files and their usage are described in detail in the manual [16]. The function `bvpsuite2` operates in the following way:

Algorithm 3: `bvpsuite2.m`: Main function of `bvpsuite2.0`

Input: problem definition, solver settings; optional: initial profile, pathfollowing settings

Output: final mesh, values on this mesh of the approximate solution, further solution information

```
                                                          // In bvpsuite2.m at line
1  Initialize ;                                                               // 39
2  Define the initial profile either from the function call, the problem definition
     or else, take the constant solution 1 ;                                  // 56
3  if problem EVP then Adjust the initial profile ;                           // 81
4  if pathfollowing active then                                               // 101
5      Call pathfollowing ;                                                   // 112
6      return last solution on path ;
7  end
8  if problem is linear then                                                  // 140
9      Call meshadaptation, solveLinearProblem or errorestimate,
         depending on solver settings, to compute an approximate solution;
10 else                                                                       // 155
11     Call meshadaptation, solveNonLinearProblem or errorestimate,
         depending on solver settings, to compute an approximate solution;
12 end
```

An initial profile is only needed for the non-linear solver. The required adjustments in

the cases of a BVP posed on a semi-infinite interval or in the case of an EVP are detailed in the discussion of the function `feval_problem` below.
An initial profile can also be passed as a function argument. Then the initial profile defined in the problem definition file will be ignored and the initial profile from the function argument will be used. This works similarly for the pathfollowing settings, which can be passed through the function argument. The settings, which are passed through the function argument, will replace the pathfollowing settings which are given in the problem definition under `'pathfollowing'`, as well as the seven settings which are set in the solver settings. For examples which show the usage of this functionality, see in Section 1.3 the first example of the Bratu equation, or the manual [16, Sec. 3.7]. This functionality was added to allow changes to the settings within scripts.
Changelog for `bvpsuite2`:

- For pathfollowing problems, the function calls the function `pathfollowing`, right after collecting all the required data about the initial mesh and profile. After the pathfollowing function returns a path, inside the function `bvpsuite2`, the path will be saved to the folder specified in the problem definition file.
- The function argument `ValuesAt` was removed, since it is not used anywhere now.
- The function argument `pathfoll` was added, to transmit pathfollowing specific data more easily.
- For eigenvalue problems that are not parameter-dependent, it is no longer needed to transmit an empty vector `[]`, as "initial value" for the parameters within the function call of `bvpsuite2`.

`feval_problem`

The function `feval_problem` is called in order to interpret the problem definition according to some specificities relevant to the execution of the solver routine. Two problem cases are handled in a special way, namely the case where the problem is

- an eigenvalue problem: the eigenvalue λ is treated as an unknown parameter. The normalization condition with respect to the L_2-norm, which fixes the values of the eigenfunction, is also introduced by adding a new function $z_{n+1}(t)$ to the problem, satisfying the equation

$$z'_{n+1}(t) = \sum_{i=1}^{n} z_i(t)^2.$$

  Two boundary conditions are also added, to fix these unknowns, namely the naturally resulting conditions

$$z_{n+1}(a) = 0 \quad \text{and} \quad z_{n+1}(b) = 1.$$

- posed on a semi-infinite interval: here the interval is transformed from $[a, \infty)$ to $[0, a]$, when $a \in \mathbb{R}^+$, by the function $\xi : [a, \infty) \to [0, a], t \mapsto \frac{1}{t}$. If $a = 0$, then the interval is partitioned into $[0, 1]$ and $[1, \infty)$. The left interval $[0, 1]$ stays the same and the right interval $[1, \infty)$ is handled just like the case above when $a \in \mathbb{R}^+$. The problem equations are computed then on both intervals, which means that the number of unknown solution components is doubled. So the double amount of boundary conditions is also needed, which is given by continuity conditions imposed on the approximations and their derivatives.

These cases can also occur at the same time. For both cases, more details on their workings can be found in [1]. The function `feval_problem` ensures to insert the correct values into the vector $\boldsymbol{F}$, as in (1.34), and the matrix $\boldsymbol{DF}$, as in (1.37). It operates in the following way:

Algorithm 4: `feval_problem.m`: Return the requested characteristic entities of the problem.

Input: problem definition, requested entity, further input
Output: requested entity of the problem

```
                                                      // In feval_problem.m at line
 1 Initialize ;                                                               // 3
 2 if interval is finite then                                                 // 53
 3     if problem is an EVP then                                              // 54
 4         Compute requested entity ;
 5     else                                                                   // 107
 6         Compute requested entity ;
 7     end
 8 else                                                                       // 114
 9     if problem is an EVP then                                              // 115
10         Compute requested entity ;
11     else                                                                   // 302
12         Compute requested entity ;
13     end
14 end
```

Note that in the case, when the problem is not an EVP and the interval is finite, then the requests can be evaluated straight away from the problem definition file. As discussed above, for semi-infinite intervals there is also always the distinction between the case where $a = 0$ and where $a > 0$.

Changelog for `feval_problem`:

- For eigenvalue problems, where the boundary values are imposed somewhere within the interval, the case `'c'` is used in the problem definition. In order to impose the boundary conditions at a and b, as explained above, in `feval_problem` the vector `'c'` is augmented by the right and left interval limits. This was not taken into account when requesting `'dBV'`. This was changed and tested successfully.
- The pathfollowing variable λ^* was added where needed.

coeffToValues

The function coeffToValues returns the values of the polynomial, as in (1.28), approximating the solution function at any points of the interval. It is used in different places in the bvpsuite2.0 routine, for instance in the errorestimate function, when the polynomial on the coarse mesh needs to be evaluated on the fine mesh, in order to compute the error estimate. Also it can be used by the user as a function to evaluate the solution at any point in the interval. In pathfollowing problems, this function is very useful to define the function PathCharData, as is shown in Section 1.2.2. It operates in a straightforward fashion in the following way:

Algorithm 5: `coeffToValues.m`: Returns function values for given function coefficients.

```
Input: polynomial coefficients, mesh on which the solution was computed,
       points on which to evaluate, which derivative
Output: function, or one of its derivatives, values at the requested points
                                                    // In coeffToValues.m at line
1 Initialize ;                                                          // 015
2 foreach requested point do                                            // 040
3     Evaluate and sum the $\Phi_{ij}$ multiplied by the coefficients ;       // 043
4     Evaluate and sum the $\Psi_{ij}^{lk}$ multiplied by the coefficients ;  // 050
5 end
```

In the function coeffToValues the formula (1.28) is evaluated.

Changelog for `coffToValues`:

- Some unused lines, such as the locally defined but unused function gauss, were removed.

initial_coefficients

The function initial_coefficients was implemented to find the coefficients of a polynomial in the Runge-Kutta basis of the initial profile provided by the user. Additionally, in the pathfollowing routine, it is used, when the mesh was adapted, to adjust the coefficients vector of the approximating polynomial to the new mesh. The function operates in the following way:

Algorithm 6: `initial_coefficients.m`: Returns the polynomial coefficients of a given function.

```
Input: problem definition, discrete mesh, function values at mesh points
Output: corresponding coefficients to the function
                                          // In inital_coefficients.m at line
1 Initialize ;                                                        // 005
2 foreach solution function do                                        // 079
3     Interpolate the values of the function profile, given at the mesh points, at
        all collocation points ;
4     foreach interval in-between two mesh points do                  // 191
5         Compute the polynomials coefficients ;
6     end
7 end
8 Transform the coefficients to the form (1.33) ;                     // 225
```

Changelog for `initial_coefficients`:

- In line 20 the condition `interval(1)==0` was added, for only in that case the semi-infinite interval would need splitting.

`solveLinearProblem`

The function `solveLinearProblem` is the solver routine for linear problems. When dealing with a linear problem, all the coefficients can be computed from the Jacobian $\boldsymbol{DF}$ evaluated at $\mathbf{0}$ and the function $\boldsymbol{F}$ evaluated at $\mathbf{0}$ gives the inhomogeneity of the system. Then, this is a closed system of equations with as many unknown coefficients in the polynomials in the Runge-Kutta basis as conditions in the problem equations, boundary conditions and continuity conditions. The function `solveLinearProblem` thus operates in the following way:

Algorithm 7: `solveLinearProblem.m`: Solver for linear problems.

```
Input: problem definition, mesh
Output: approximate solution
                                          // In solveLinearProblem.m at line
1 Initialize ;                                                        // 003
2 Compute residual $\boldsymbol{F}(\mathbf{0})$ with functionFDF ;    // 070
3 Compute system matrix $\boldsymbol{DF}(\mathbf{0})$ with functionFDF ; // 075
4 $\boldsymbol{c} = \boldsymbol{DF}(\mathbf{0})^{-1}\boldsymbol{F}(\mathbf{0})$ ; // 080
5 Transcribe the coefficients $\boldsymbol{c}$ into the vectors $Y$ and $Z$ as in (1.33) and $p$ for
    the parameters;                                                   // 082
6 Evaluate the polynomials at the mesh points ;                       // 097
```

To this function the solver settings are not passed. This is due to the fact that here a closed system of equations is solved once, in contrast to the case of non-linear problems, where an iterative solver is employed. Here the errors can only be improved upon by changing the mesh. The error for linear problems not only consist of rounding errors in the floating point arithmetic, but also errors due to the continuity condition imposed on the approximate solution and its derivatives, that needs to be satisfied as well as the problem equations and the boundary conditions. For linear problems with rapid solution changes, this may cause errors, that may be handled by adapting the mesh used in the approximation of the solution, see for example the linear problem in Section 2.2.

Changelog for `solveLinearProblem`:

- The vector $\boldsymbol{F}(\mathbf{0})$ and the matrix $D\boldsymbol{F}(\mathbf{0})$ are computed with the function `functionFDF` instead of, as before, directly in the function itself. The main advantage is that it makes the function a lot shorter.

solveNonLinearProblem

The function `solveNonLinearProblem` prepares the problem data to be passed to the function `solve_nonlinear_sys`, which comprises the main solver routine of `bvpsuite2.0`. The function `solveNonLinearProblem` operates in the following way:

Algorithm 8: `solveNonLinearProblem.m`: Prepares data for the approximation iteration.

```
Input: problem definition, solver settings
Output: approximate solution $s$
                                          // In solveNonLinearProblem.m at line
1 Initialize ;                                                            // 003
2 $\boldsymbol{c}$ = solve_nonlinear_sys ;                                // 061
3 Transcribe the coefficients $\boldsymbol{c}$ into the vectors $Y$ and $Z$ as in (1.33) ;   // 070
4 Evaluate the polynomials at the mesh points ;                           // 087
```

This function helps to separate all the preparation and processing of the data from the function `solve_nonlinear_sys`, which contains the main solver routine.

Changelog for `solveNonLinearProblem`:

- The local functions `F` and `DF` have been moved to the function `functionFDF`. They were not kept as local functions. They are no longer transmitted to `solve_nonlinear_sys` either.
- The `predictor` variable, containing the required values to compute the last line of $D\boldsymbol{F}$ in case of pathfollowing, was added to the function.
- Some unused lines were removed, as for instance the computation of the collocation points is handled by the function `getStandardCollocationPoints` now, instead of the local functions `gauss` and `lobatto`, which were removed.

`solve_nonlinear_sys`

The function `solve_nonlinear_sys` is the function performing the main solver iteration, in order to get an approximate solution starting from a given initial guess. Before trying to reach the required tolerances by the user, inside the local function `determine_position2` a step of the fast frozen Newton method, starting at the initial profile that was transmitted to the function, is performed. Depending on the convergence exhibited in this first step, one of the three implemented algorithms, the fast frozen Newon method, the predictor-corrector line search or the trust-region method, is chosen. This algorithm is then performed in order to satisfy the tolerances, whereby if the convergence in later steps improves or deteriorates, an other one of the three algorithms may be chosen in order to proceed. The function thus operates in the following way:

Algorithm 9: `solve_nonlinear_sys.m`: Iteration scheme to approximate solution.

Input: initial profile, problem definition, solver settings

Output: approximate solution

```
                                              // In solve_nonlinear_sys.m at line
 1 Initialize ;                                                              // 008
 2 Call determine_position2 to choose iterative algorithm ;                  // 062
 3 while 1 do                                                                // 092
 4     switch iterative algorithm do                                         // 108
 5         case Fast Frozen Newton do                                        // 112
 6             Compute boolean Up_Jac to update Jacobian ;                   // 117
 7             if Up_Jac== 1 then Update the Jacobian ;                      // 126
 8             Perform a step of the fast frozen Newton method ;
 9             if Approximation improved then                                // 180
10                 if Tolerances are satisfied then return;                  // 191
11                 Keep trying with the fast frozen Newton method ;
12             else if Up_Jac== 0 then                                       // 216
13                 Keep trying with the fast frozen Newton method ;
14             else                                                          // 226
15                 Change to Predictor-Corrector Line Search ;
16             end
17         case Predictor-Corrector Line Search do                           // 250
18             Introduce and improve on a damping parameter $\lambda$ in the fast
                 frozen Newton method ;
19             if $\lambda < \lambda_{min}$ then Switch to trust region method;  // 304
20             if Approximation improved then                                // 339
21                 Try the fast frozen Newton method next ;
22             else                                                          // 341
23                 Keep trying with the predictor-corrector line search ;
24             end
25             if Tolerances are satisfied then return;                      // 363
26         case Trust Region Method do                                       // 463
27             if Pathfollowing problem then return;                         // 466
28             One of the two built-in MATLAB functions lsqnonlin or fsolve is
                 used ;
29             Call determine_position2 to choose the algorithm with which to
                 proceed ;                                                   // 547
30             if Tolerances are satisfied then return;                      // 550
31     end
32 end
```

The details concerning the fast frozen Newton method and also the predictor-corrector

line search can be found in [2].
The domain of converge increases from the fast frozen Newton method to the predictor-corrector line search to the trust region method, but just as well the computational effort does.
The tolerances are satisfied, whenever the norm of the Newton increment gets small enough. More precisely, the Newton step is performed through the formula

$$\boldsymbol{x}_{new} := \boldsymbol{x}_{old} + \Delta\boldsymbol{x}_{old},$$

where $\boldsymbol{x}_{old}$ is the current vector of coefficients of polynomials in the Runge-Kutta basis as in (1.28) and $\Delta\boldsymbol{x}_{old}$ is the Newton increment which is a solution of the equation

$$D\boldsymbol{F}(\boldsymbol{x}_{old})\Delta\boldsymbol{x}_{old} = -\boldsymbol{F}(\boldsymbol{x}_{old}).$$

Then, the tolerances are satisfied when

$$Tol := \frac{\max(\Delta\boldsymbol{x}_{old})}{aTol + rTol \max_{j\in\{1,\ldots,\text{length}(\boldsymbol{x}_{old})\}}\left|\boldsymbol{x}_{new,j}\right|} < 1,$$

where $aTol$ is the absolute tolerance and $rTol$ is the relative tolerance, both prescribed by the user.
Changelog for `solve_nonlinear_sys`:

- The `predictor` variable was added where needed.
- In line 277, at the beginning of the predictor-corrector line search algorithm part, a missing square was added, which is present in the corresponding formula in [2].

errorestimate

The function `errorestimate` is called either from `bvpsuite2` or `meshadaptation`, when in the solver settings the error estimate or the mesh adaptation – which uses the information from the error estimate to adapt the mesh – are activated. It operates in the following way:

Algorithm 10: `errorestimate.m`: Compute an estimate for the error in the approximation.

Input: approximation, problem definition, solver settings

Output: approximation with error estimate, approximation on fine mesh with error estimate

```
                                              // In errorestimate.m at line
1  Initialize ;                                                          // 3
2  Add a point in-between each two mesh points of the original mesh ;  // 36
3  if problem is linear then                                             // 39
4      Call solveLinearProblem and compute an approximation on the fine
       mesh;
5  else                                                                  // 41
6      if Pathfollowing problem then                                     // 42
7          Adjust the coefficients vector to the refined mesh ;
8          Set the flag in predictor to keep $\lambda^*$ fixed ;         // 70
9      end
10     Call solveNonLinearProblem, with the available approximation as initial
       profile, to compute an approximation on the fine mesh ;
11 end
12 Compute the estimates for the error with the two approximations on
   different meshes and the formulas (2.1) and (2.2) ;                   // 84
```

Let m be the number of collocation points used to approximate the solution. The approximation $p(t)$ to the problem is computed on the discrete mesh $\mathcal{M}$ with mesh intervals h_i and the approximation $p_2(t)$ on the finer mesh $\mathcal{M}_2$ with mesh intervals $\frac{h_i}{2}$. Assume that a function $e(t)$ exists such that for the errors $\delta(t) := p(t) - z(t)$ and $\delta_2(t) := p_2(t) - z(t)$ it holds

$$\begin{aligned} \delta(t) &= e(t)h_i^m + \mathcal{O}(h_i^{m+1}) \quad \text{and} \\ \delta_2(t) &= e(t)\frac{h_i^m}{2^m} + \mathcal{O}(h_i^{m+1}). \end{aligned}$$

Then, $\delta(t) - 2^m\delta_2(t) = \mathcal{O}(h_i^{m+1})$ and when adding $2^m(p(t) - p(t))$ we obtain

$$\begin{aligned} & p(t) - z(t) + 2^m(p(t) - p(t)) - 2^m(p_2(t) - z(t)) = \mathcal{O}(h_i^{m+1}) \\ \Leftrightarrow \quad & (1 - 2^m)(p(t) - z(t)) - 2^m(p_2(t) - p(t)) = \mathcal{O}(h_i^{m+1}). \end{aligned}$$

This leads to the definition of the asymptotically correct estimate of the error $\delta(t)$

$$\epsilon(t) := \frac{2^m}{1 - 2^m}(p_2(t) - p(t)). \tag{2.1}$$

When instead $p_2(t) - p_2(t)$ is added above, then an asymptotically correct estimate of the error $\delta_2(t)$ can be defined as

$$\epsilon_2(t) := \frac{1}{1 - 2^m}(p_2(t) - p(t)). \tag{2.2}$$

These formulas are used to compute an estimate of the error, which may be quite accurate if the mesh is fine enough.
In the case of pathfollowing, before computing the approximation on the fine mesh, the vector of coefficients of the polynomials in the Runge-Kutta Basis, as in (1.28), is adjusted to the fine mesh. Starting at the approximation on the coarse mesh, the approximation on the fine mesh is computed. In this computation, the value of λ^* is kept fixed. The reason behind that is because by changing the coefficients to the fine mesh, the coefficients of the tangent and the initial point from where the pathfollowing step is started, would need to be adjusted too, in order to compute (1.36). There was no satisfactory way found to adjust these vectors to the fine mesh and to still have the orthogonality condition satisfied to the same extent as before the adjustment. Therefore, the value λ^* is kept fixed, by setting the flag in the `predictor` to 1. This setting will make the last row of $\boldsymbol{DF}$ be all zeros except for the entry in the downright corner, which is 1. In this way the matrix is still non-singular. Also the last entry of the vector $\boldsymbol{F}$ is set to 0. These two settings ensure that there is always a 0-entry at the position of the pathfollowing parameter λ^* in the Newton increment. Thus, the value of λ^* is kept fixed.
Changelog for `errorestimate`:

- A `try-catch` statement was added before starting the computations on the fine grid, in order to adjust the coefficients of the approximation on the coarse mesh to the fine mesh, if the problem is a pathfollowing problem. In the pseudo-code above, it is written as an `if` statement, in order to keep the pseudo-code simple.

`meshadaptation`

The function `meshadaptation` is called by `bvpsuite` or in the case of a pathfollowing problem by `pathfollowing`, when searching for an approximation of the solution of a BVP. The function adapts the mesh in two ways, namely by

- moving mesh points: depending on the residual given by the `computeResidual` function, which is discussed below, the mesh density ρ will be updated by the formula

$$\rho_{new}(t) = \rho(t) \cdot M(res_n(t))^{\frac{1}{k}},$$

  where M is the number of intervals, $res_n(t)$ is the norm of the residual of all equations at the time t and $k := 2.75 \cdot m$ is defined at the beginning of the function, where m is the number of collocation points. The new mesh density is then smoothed out and normalized, then applied to the old mesh $\mathcal{M}_{old}$ to form a new mesh $\mathcal{M}_{new}$.
- adding mesh points: the new number of mesh points is computed by the formula

$$N_{new} = \left\lceil (1 + s_N) \cdot N \cdot \max_{i \in \{1,\dots,n\}} \frac{m_{err,i}}{(s_\sigma \cdot aTol)^{\frac{1}{m+1}}} \right\rceil, \tag{2.3}$$

  where n is the number of solution components, m is the number of collocation points used, $aTol$ is the absolute required error tolerance, s_N and s_σ are two constants set at the beginning of the function to 0 and 0.9 respectively, and $m_{err,i}$ is the maximal value of the error estimate obtained for each solution component $i \in \{1, ..., n\}$. The new points are added in accordance with the mesh density, which is smoothed out and normalized again before applying it to the old mesh $\mathcal{M}_{old}$ with N mesh points, to form a new mesh $\mathcal{M}_{new}$ with N_{new} mesh points.

The adaptation process continues until the error estimates satisfy the tolerances prescribed by the user in the solver settings file under `'absTolMeshAdaptation'` and `'relTolMeshAdaptation'`.

The function `meshadaptation` thus operates in the following way:

Algorithm 11: `meshadaptation.m`: Adapt the mesh until termination condition is met.

Input: problem definition, solver settings

Output: approximation satisfying the tolerance requirements

```
                                                          // In meshadaptation.m at line
1  Initialize ;                                                            //  22
2  Initialize the update-mode u_m = 1 ;                                    //  70
3  Ncompare = $10^8$ ;                                                     //  75
4  foreach iteration, until maximal iteration number is reached do         // 105
5      Compute a solution on mesh with linear or non-linear solver ;       // 114
6      Compute the error estimate ;                                        // 147
7      if tolerances are satisfied then break;                             // 189
8      if u_m == 2 then                                                    // 219
9          if mesh density has not been updated more than twice then       // 220
10             u_m = 1 ;
11         else                                                            // 223
12             Update the number of mesh points with formula (2.3) ;
13         end
14     else if u_m == 1 then                                               // 229
15         Ncompare = Ncompare_old ;
16         Compute Ncompare with formula (2.3) ;
17         if Ncompare > (1−minimprove) Ncompare_old then                  // 237
18             u_m = 2 ;
19     end
20     if u_m == 1 then Compute residual with computeResidual ;            // 270
21     if u_m == 1 then Compute new mesh density with LPFilter ;           // 351
22     Smoothen the new mesh density with bcTDFlogV4, normalize it and then
        apply it to the mesh and its new mesh points ;                     // 360
23     In case of pathfollowing, update the coefficients vector to the new mesh
        and set the flag to keep $\lambda^*$ fixed ;                       // 371
24 end
```

The functions LPFilter and bcTDFlogV4 are short local functions, that are solely used here.
The tolerances are satisfied in the mesh adaptation procedure, when

$$Tol := \max_{i\in\{1,\dots,n\}} \frac{\max \boldsymbol{errest}_i}{aTol + rTol \, \max_{j\in\{1,\dots,\text{length}(\boldsymbol{y}_i)\}} \left|\boldsymbol{y}_{i,j}\right|} < 1,$$

where n is the number of solution components, $errest_i$ the maximal value of the error estimate for the i-th component, $aTol$ the absolute tolerances and $rTol$ the relative tolerance, both set by the user in the solver settings, and $\boldsymbol{y}_i$ are the approximate values of the i-th solution component.
Similarly to the errorestimate function, when the mesh is adapted, the coefficient vector for the initial solution of the next iteration process is adjusted to the new mesh. Here, after the approximation on the initial mesh, the value of λ^* is kept fixed.
Changelog for meshadaptation:

- At the very end of the mesh adaptation loop, the coefficients of the approximation on the previous mesh are adjusted to the new mesh, if the problem is a pathfollowing problem.
- Some minor adjustments were made, to improve readability of the code, i.e. the obsolete transformation of the approximation for problems on semi-infinite intervals, which is now performed in the function bvpsuite2, was removed. Also in the if-else statement after the computation of the error estimate, the if-else statement checking whether or not the tolerances are satisfied, was put right after the error estimation, in order to avoid repeating it in both cases of update-mode.

computeResidual

The function computeResidual is used during the mesh adaptation procedure, in order to evaluate the implicit problem equations, as given in (1.26), at specific points in the interval. It will return the residual of the evaluation of the equations whose profile is used in the mesh adaptation procedure.

Algorithm 12: computeResidual.m: Evaluate the problem equations at the approximate solution.

Input: problem definition, coefficients, points on which to evaluate
Output: residual of the problem equations

```
                                              // In computeResidual.m at line
1 Initialize ;                                                          // 006
2 Compute all the approximate solutions and its derivatives, using
    coeffToValues, at the requested points ;                            // 010
3 foreach requested point do                                            // 015
4   | Compute residual using feval_problem ;
5 end
```

Changelog for `computeResidual`:

- The pathfollowing variable λ^* was added to the function.

computeEVPStart

The function `computeEVPStart.m` can be called before calling `bvpsuite2`, in order to compute good initial guesses for a linear eigenvalue problem of the form

$$\boldsymbol{f}(t, \boldsymbol{z}^{(1)}(t), \ldots, \boldsymbol{z}^{(\hat{l})}(t), \boldsymbol{p}) = \lambda \boldsymbol{z}(t), \tag{2.4}$$

with linear boundary conditions. The way this function works, is that it produces the matrix $D\boldsymbol{F}(\mathbf{0})$, as in (1.37), for the left hand side of the equation and for the right hand side on a discrete mesh containing only a few points. The results can then be used as input for the built-in MATLAB function `eig`. More explanations to this function can be found in [1]. The function operates in the following way:

```
Algorithm 13: computeEVPStart.m: Returns initial guesses for an eigenvalue problem.
  Input: problem definition, solution profile
  Output: initial guesses
                                                   // In computeEVPStart.m at line
1 Initialize ;                                                                // 003
2 Compute the collocation matrices L and R, which represent the LHS and the
    RHS of (2.4), respectively ;                                              // 072
3 The function eig is used to compute the initial eigenvalues and coefficients of
    the eigenfunctions of the problem ;                                       // 226
4 The initial profile struct-variables are prepared ;                         // 238
```

An example run of this function is found in the following Section 2.2.
Changelog for `computeEVPStart`:

- The pathfollowing variable λ^* was added where needed.

Remaining functions

The remaining functions, which are shorter and uncomplicated in comparison to the ones mentioned above, are

- `template_bvp`: template for the problem definition, see Section 1.2.2 for changes to that file;
- `default_settings`: template for the solver settings, see Section 1.2.2 for changes to that file;
- `dispDebug`: tool for debugging purposes;
- `getStandardCollocationPoints`: will return the required collocation points in $[0, 1]$;
- `trafo`: provides the standard transformation when dealing with a semi-infinite inter-

val;

- `higherchainrule`: is used in `trafo` to compute the required derivatives of the transformation.

The functions which were already discussed in the previous chapter about pathfollowing are

- `pathfollowing`: main function of the pathfollowing module, and
- `functionFDF`: function to compute $\boldsymbol{F}$, as in (1.34), and $D\boldsymbol{F}$, as in (1.37), which were local functions in the function `solveNonLinearProblem`, but were taken out since they are needed in the pathfollowing module.

Finally, functions that are still at the stage of being developed are

- `trafo_expmt`: this function comprises a template for a user-defined transformation for problems posed on semi-infinite intervals;
- `feval_problem_2`: this function is built in the same way as `feval_problem`, with the difference that it handles a user-defined transformation, given in `trafo_expmt`, for problems posed on semi-infinite intervals. For this function some special cases have not been worked out yet.

This concludes the discussion of the functions that are implemented and present in the `bvpsuite2.0` package.
In the next section, the focus will lie on demonstrating the capabilities of the package by computing an example per module.

2.2 Modules of `bvpsuite2.0`

The seven modules of `bvpsuite2.0` shall be presented in this section. Each module can handle a special type of problems.
The only two cases that are mutually exclusive are when the problem is linear and thus the solution can be approximated with the linear solver, or else the nonlinear solver is chosen and then the solution is iteratively approximated by the nonlinear solver. All the other cases can occur at the same time, with some restrictions concerning mainly boundary conditions in the case of problems posed on semi-infinite intervals.
One example will be presented for each problem case. These examples can also be found in the manual for `bvpsuite2.0` ([16]) with the corresponding lines of code to execute these runs. The examples were chosen and implemented in collaboration with Aron Sass, and built upon the prerequisite work of Stefan Wurm.
The seven modules of `bvpsuite2.0` are

These are the current problem types `bvpsuite2.0` is programmed to handle.
In this work, these examples played a crucial role in testing all `bvpsuite2.0` modules after some changes were made during the implementation of the pathfollowing module.

Linear problems

The following example was taken from [1, Sec. 2.3].
Consider the linear, singularly perturbed BVP, with $\varepsilon = 10^{-4}$,

$$\begin{aligned} &\varepsilon z''(t) + z'(t) - (1+\varepsilon)z(t) = 0, \quad t \in [-1,1]\,, \\ &z(-1) = 1 + \mathrm{e}^{-2}, \quad z(1) = 1 + \mathrm{e}^{\frac{-2(1+\varepsilon)}{\varepsilon}}. \end{aligned} \tag{2.5}$$

For the approximation of the solution to this BVP a starting mesh with 201 equidistant mesh points and Gauss collocation with 4 collocation points in-between each two mesh points was chosen. Two runs were carried out. In the first run the relative and absolute tolerances for the mesh adaptation were set to 10^{-9}. The tolerances on the non-linear solver do not need to be set, since this is a linear problem. The mesh is adapted, but the number of mesh points is kept at 201 in the final mesh. The results are displayed in Figure 2.2.

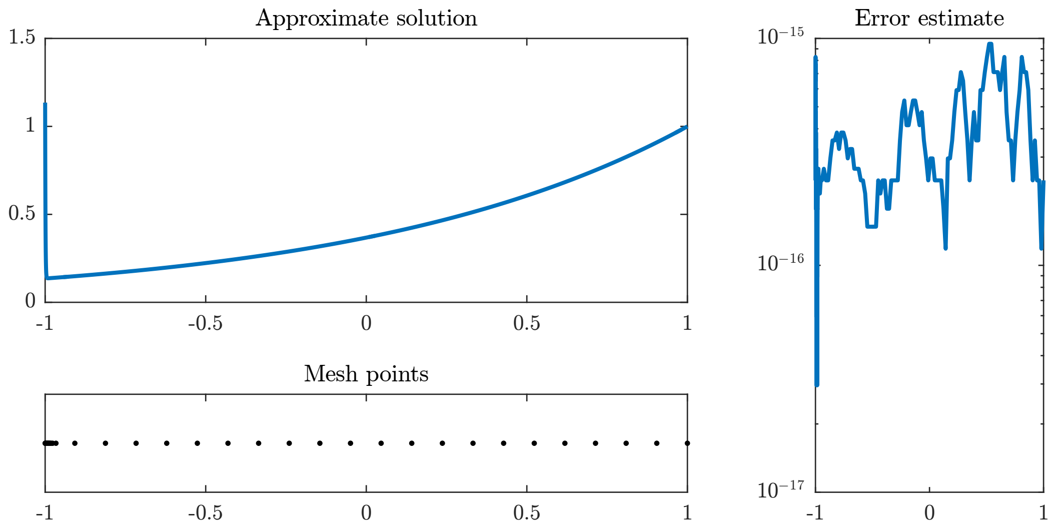


Figure 2.2: Linear Problem: Approximated solution, mesh points distribution and error estimate computed with mesh adaptation.

In the second run, the mesh adaptation was turned off. The result of this run are displayed in Figure 2.3.
The error in the second around $t = -1$ is more then 10^{10} times as high as the error in the first run, although the number of mesh points in the final mesh in the two runs is

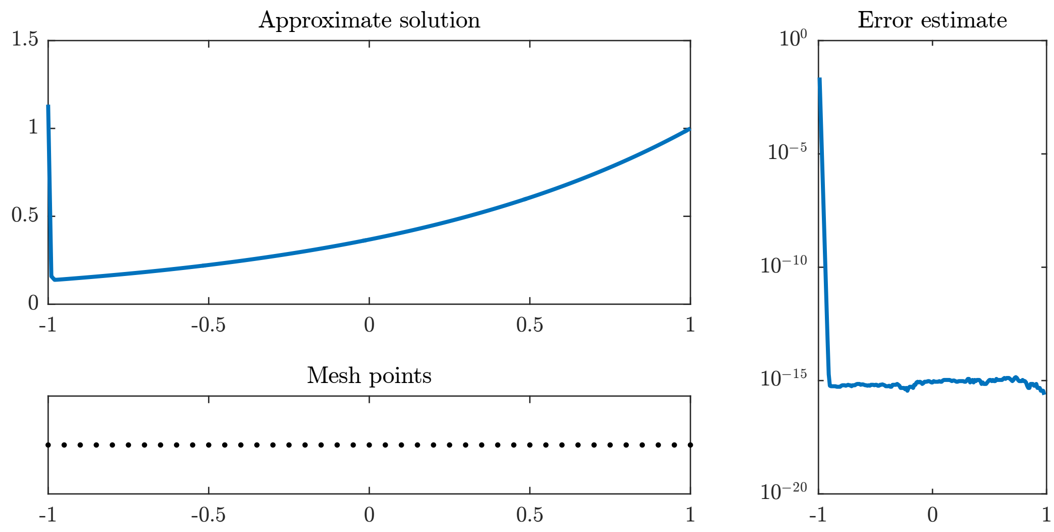


Figure 2.3: Linear Problem: Approximate solution, mesh points distribution and error estimate computed without mesh adaptation.

the same. This goes to show that, the mesh adaptation can be a very useful tool for some problems.

Nonlinear problems

The following example was taken from [15].
Consider the nonlinear singular BVP, where $\nu = \frac{1}{3}$, $\mu = 9$ and $\gamma = 1000$,

$$\begin{aligned} &z_1''(t) + \frac{3}{t} z_1'(t) = -\mu^2 z_2(t) - 2\gamma + z_1(t) z_2(t), \\ &z_2''(t) + \frac{3}{t} z_2'(t) = \mu^2 z_1(t) - \frac{1}{2} z_1^2(t), \quad t \in (0,1]\,, \\ &z_1'(0) = 0, \qquad z_2'(0) = 0, \\ &z_1(1) = 0, \qquad z_2'(1) + (1-\nu) z_2(1) = 0. \end{aligned}$$

The division by 0 can lead to `NaN` values in MATLAB. This will not happen during the solver iteration for this problem, since in the above equations the singularity occurs at one of the two interval boundaries. The equations are only evaluated at the collocation points in-between the points of the discrete mesh, which would start at 0 and end at 1. Thus, the problem can be entered as written above in the problem definition file. If the singularity in a problem is of a higher order α, this is still not a problem, but sometimes for convergence purposes, it might be advantageous to multiply the equation by t^α.
The approximation to this BVP was found using a starting mesh with 101 equidistant mesh points and Gauss collocation with 3 collocation points in-between each two mesh points. The relative and absolute tolerances for the non-linear solver were set to 10^{-12}

and to 10^{-9} for the absolute and relative tolerance of the mesh adaptation. The initial approximation was chosen to be the constant function 1 for both approximations. The final mesh still contains 101 mesh points and the points were not displaced. The approximate solution and error estimate are calculated and displayed in Figure 2.4.

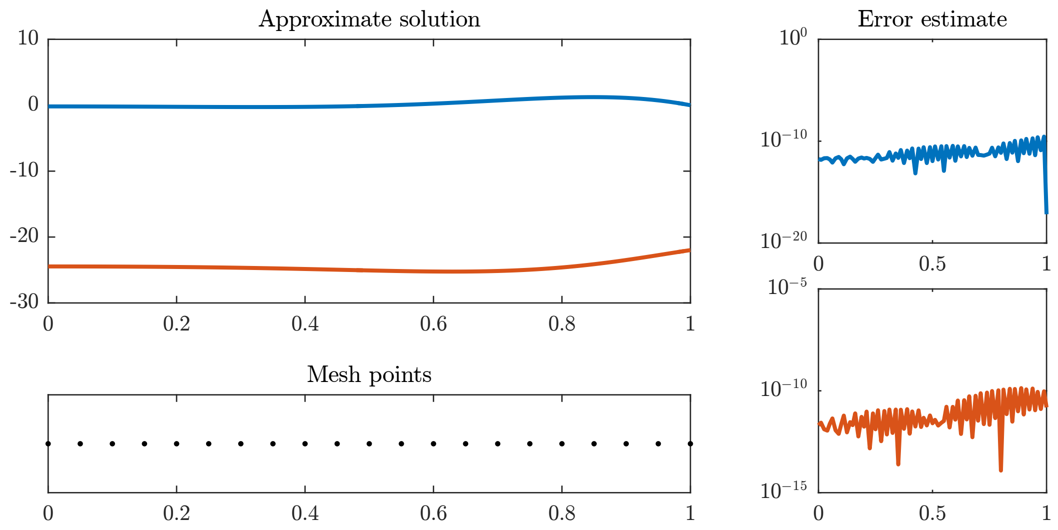


Figure 2.4: Non-linear Problem: Approximate solution, mesh points distribution and error estimate.

Parameter dependent problems

The following example was taken from [1, Sec. 2.4].
Consider the BVP

$$
\begin{aligned}
z_1'(t) &= -\frac{p}{1-p}\frac{\sqrt{t}-z_1(t)}{t-z_1(t)}, \\
z_2'(t) &= z_1(t), \quad t \in [0,1], \\
z_1(0) &= 0, \quad z_2(0) = 0, \quad z_2(1) = p.
\end{aligned}
$$

We chose to compute the problem with a starting mesh with 101 equidistant mesh points and Gauss collocation with 3 collocation points in-between each two mesh points. The tolerances were set to 10^{-6} for the absolute and relative tolerance of the non-linear solver and to 10^{-4} for the absolute and relative tolerance of the mesh adaptation. As initial profile the constant 1 function was chosen and the value 1 for the parameter.
Note that in this example, if p is initialized with 1 or whenever it reaches the value 1, a division by 0 would occur and the code would eventually throw an error. Since the initial value 1 was chosen for the parameter, this problem is remedied by multiplying the first equation by $(1-p)$ in the problem definition and thus there would not be any division by 0 occurring during the procedure.

The computation finishes after some mesh adaptation with a final mesh with 201 points. The approximate solution, error estimate and the mesh points distribution plots are displayed in Figure 2.5.

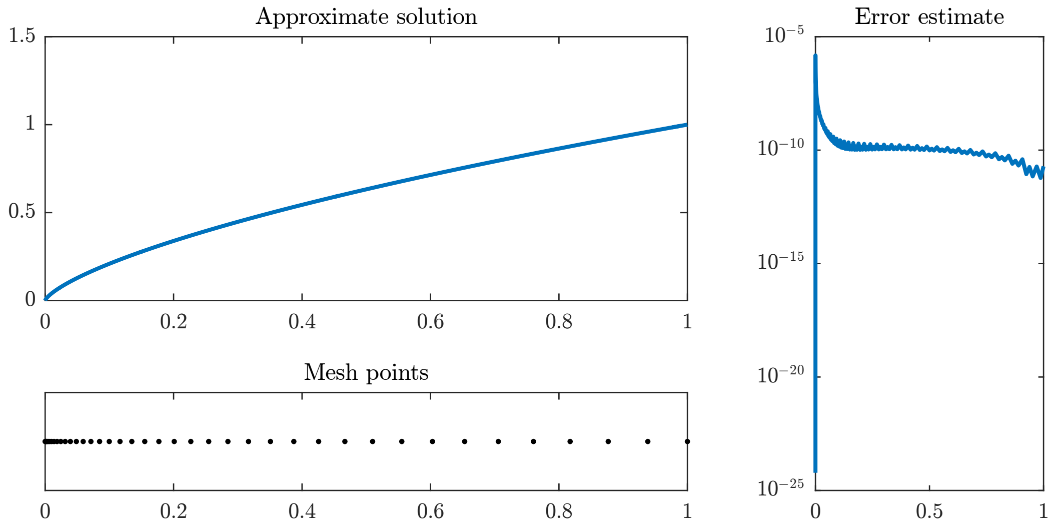


Figure 2.5: Parameter-dependent Problem: Approximated solution, mesh points distribution and error estimate.

Eigenvalue problems

The following problem was taken from [1, Sec. 4.2].
Consider the EVP

$$-z''(t) + \frac{c}{t^2} z(t) = \lambda z(t), \quad t \in (0, \pi), \quad c > 0,$$
$$z(0) = z(\pi) = 0.$$

The soluton to this EVP was approximated using a starting mesh with 51 equidistant mesh points and Gauss collocation with 3 collocation points in-between each two mesh points. The tolerances are set to 10^{-12} for the absolute and relative tolerance of the non-linear solver and to 10^{-9} for the absolute and relative tolerance of the mesh adaptation. The mesh is then adapted to a final mesh containing 227 points. The result of this computation is displayed in Figure 2.6.
Since there is an indefinite number of solutions which solve the EVP, the function `computeEVPStart` was implemented in `bvpsuite2.0` to find good initial approximations, as explained in Section 2.1. With this function, seven initial profiles were found and starting from there, approximations were computed. The tolerances were once kept at 10^{-12} for the absolute and relative tolerances of the non-linear solver and 10^{-9} for the absolute and relative tolerances of the mesh adaptation and in an other run adjusted to 10^{-9} and 10^{-6}. The results of the first run are displayed in Figure 2.7.

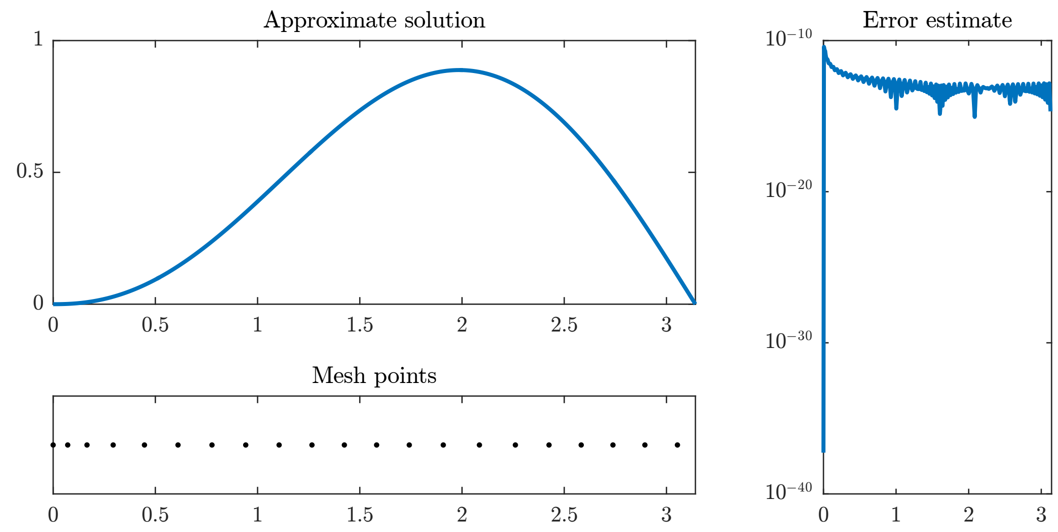


Figure 2.6: Eigenvalue Problem: Approximated solution, mesh points distribution and error estimate, initial mesh with 51 points, final mesh with 227 points.

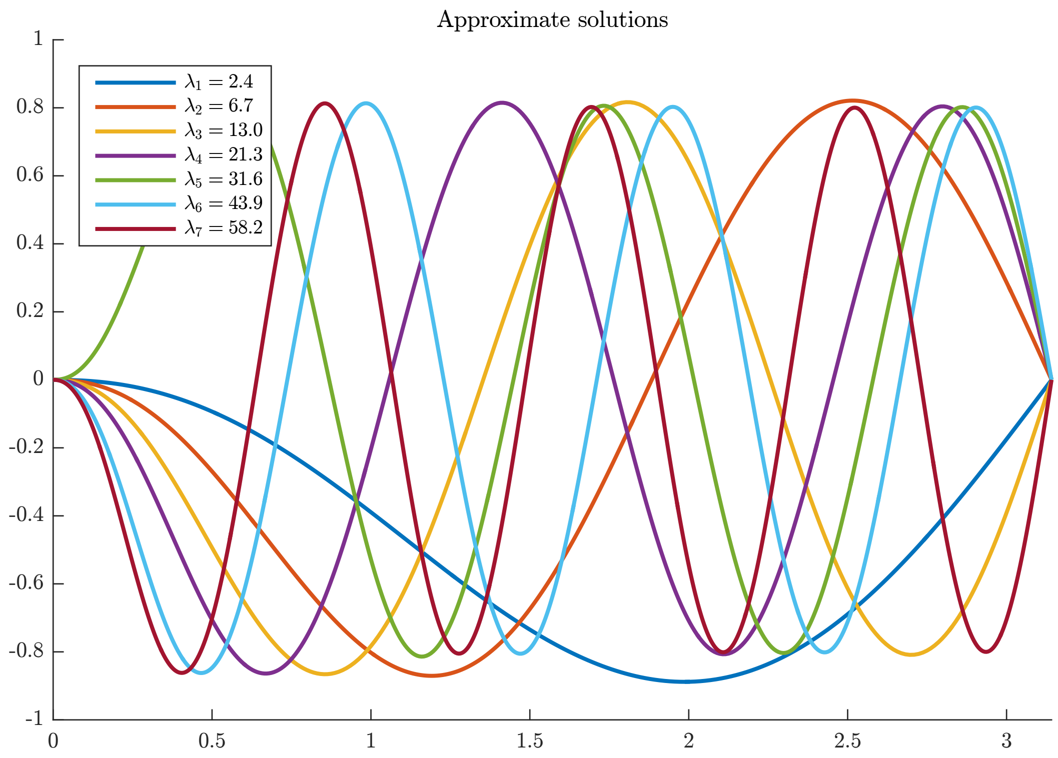


Figure 2.7: Eigenvalue Problem: Approximated solutions, computed by starting at the intial profiles given by `computeEVPStart`.

The values of the initially provided eigenvalues by `computeEVPStart` were accurate for smaller eigenvalues, but less for bigger ones, as can be seen in the table below.

	Inital EV	Final EV	Initial - final EV
λ_1	2.417106	2.417106	5.9977e-08
λ_2	6.723654	6.723653	6.4377e-07
λ_3	13.027504	13.027501	3.5746e-06
λ_4	21.330745	21.330728	1.7108e-05
λ_5	31.633815	31.633736	7.8923e-05
λ_6	43.936988	43.936647	3.4136e-04
λ_7	58.240947	58.239508	1.4392e-03

The results in both runs were very similar, the main difference was in the computation time and the number of mesh points.

		λ_1	λ_2	λ_3	λ_4	λ_5	λ_6	λ_7
Tolerances 10^{-12} and 10^{-9}	Time (in s)	87	161	133	232	262	279	371
	# mesh points	227	417	333	455	561	664	791
Tolerances 10^{-9} and 10^{-6}	Time (in s)	10	9	16	24	40	47	41
	# mesh points	51	51	51	66	74	84	93

The error estimate of the first run is displayed in Figure 2.8 and the error estimate of the second run in Figure 2.9.

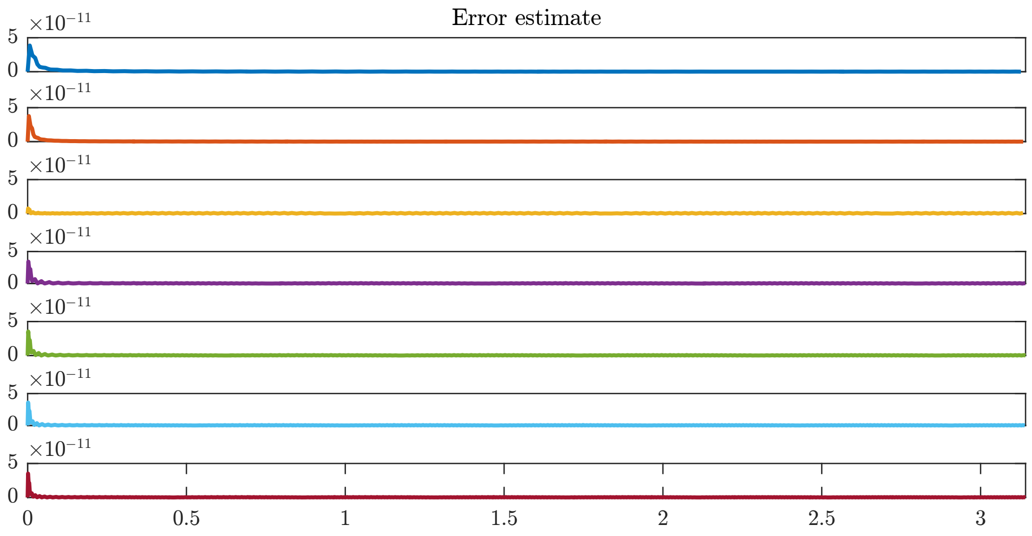


Figure 2.8: Eigenvalue Problem: Error estimate for the run with tolerances 10^{-12} for the non-linear solver and 10^{-9} for the mesh adaptation.

Only the second run with the less strict tolerances was put into the manual [16], mainly because the computation time for the stricter tolerances was higher, as can be seen in the table above.

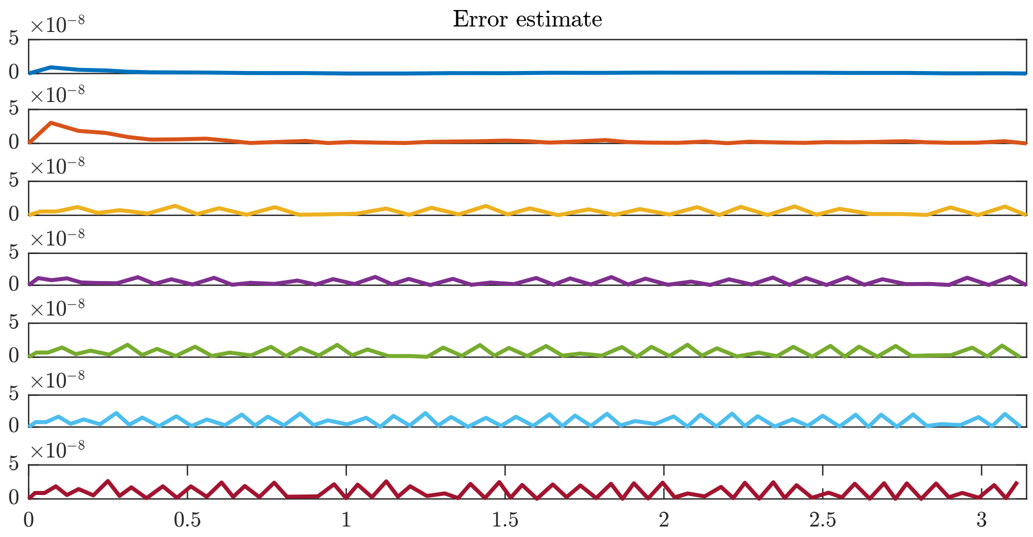


Figure 2.9: Eigenvalue Problem: Error estimate for the run with tolerances 10^{-9} for the non-linear solver and 10^{-6} for the mesh adaptation.

Index-1 differential algebraic equations

The following example was taken from [1, Sec. 6.1].
Consider the DAE, where $J = \frac{1}{2}$ and $\rho = 3$,

$$\begin{aligned}
&z_1'(t) - z_2(t)z_3(t) = 0,\\
&z_2'(t) - z_3(t) + 1 = 0,\\
&z_1(t) - \frac{J^2}{z_3(t)} - z_2(t) = 0, \qquad \text{for} \qquad t \in [0, 10.3]\,,\\
&z_1(0) - \frac{J^2}{\rho} - \rho = 0, \qquad \text{and} \qquad z_1(10.3) - \frac{J^2}{\rho} - \rho = 0,
\end{aligned}$$

As initial guess for the non-linear solver, the constant functions $\tilde{z}_1 = 1.25$, $\tilde{z}_2 = 0$ and $\tilde{z}_3 = 1$ are chosen. These satisfy the equations, but not the boundary conditions.
The solution to this BVP was approximated using a starting mesh with 51 equidistant mesh points, a uniform collocation method with 3 collocation points in-between each two mesh points, and with mesh adaptation enabled. The tolerances were set to 10^{-9} for the absolute and relative tolerance of the non-linear solver and to 10^{-6} for the absolute and relative tolerance of the mesh adaptation. The mesh is then adapted to a final mesh containing 154 points. The result of this computation is displayed in Figure 2.10.

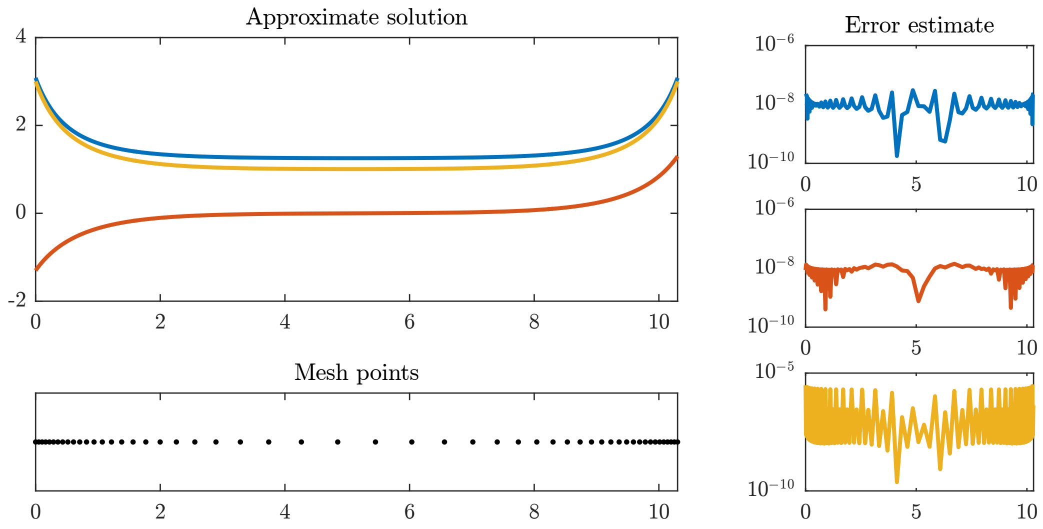


Figure 2.10: Eigenvalue Problem: Approximated solution, mesh points distribution and error estimate, initial mesh with 51 points, final mesh with 154 points.

Problems posed on a semi-infinite interval

The following example was taken from [1, Sec. 3.4].
Consider the singular BVP

$$\begin{aligned} &z''(t) + \frac{2}{t}z'(t) - 4(z(t)+1)z(t)(z(t)-0.1) = 0, \quad t \in (0,\infty), \\ &z'(0) = 0, \quad z(\infty) = 0.1. \end{aligned} \tag{2.6}$$

Following what has been done in Stefan Wurms work, the initial approximation that was used is

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

The approximation of the solution to this BVP was found using a starting mesh with 51 equidistant mesh points and Gauss collocation using 5 collocation points in-between each two mesh points. The tolerances were set to 10^{-10} for the absolute and relative tolerance of the non-linear solver and to 10^{-9} for the absolute and relative tolerance of the mesh adaptation. The tolerances are satisfied without any mesh adaptation. The results are displayed in Figure 2.11.

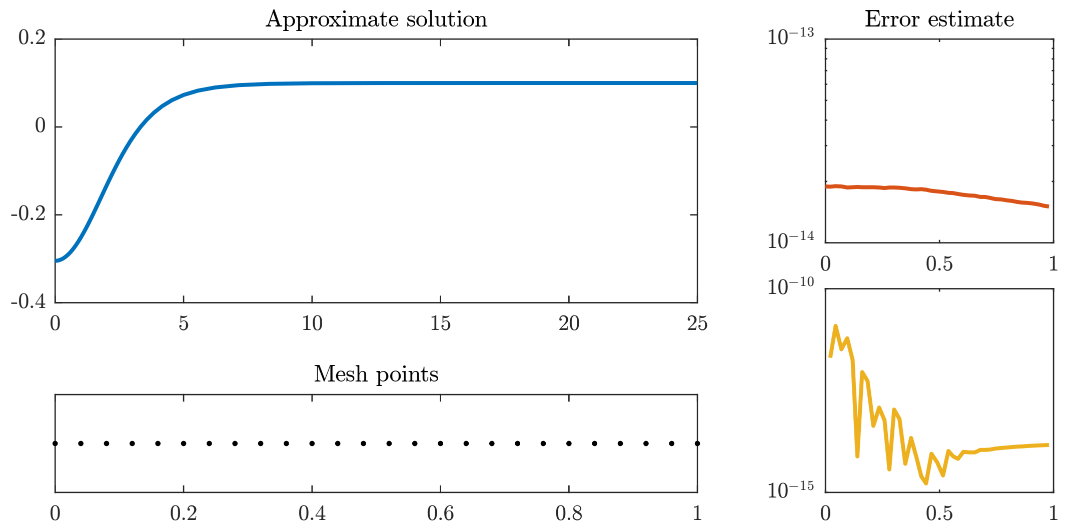


Figure 2.11: Problem posed on a semi-infinite interval: Approximated solution, mesh points distribution and error estimate.

Pathfollowing problems

The problem used as an example here is taken from [14].
Consider the BVP

$$
\begin{aligned}
&t^2 z^{(3)}(t) - tz''(t) + z'(t) - t^3 p - \lambda^* \left(tz'(t)^2 - tz(t)z''(t) + z(t)z'(t)\right) = 0,\\
&z(0) = z'(0) = 0, \quad z(1) = 0 \quad \text{and} \quad z'(1) = 1.
\end{aligned}
$$

As an initial profile, the constant function equal to 1 is chosen and for the parameter p the initial value 8 is chosen. The parameter λ^* is varied starting from 0 and with a starting step-length of 1, meaning the pathfollowing parameter will grow in positive direction with step-length 1 in the first step. During the pathfollowing, the evolution of the parameter p is observed. This is assured by the local function

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

in the problem definition file. The evolution is followed until the value -40 is reached, in order to reproduce the reference figure, as published in [14, fig. 7]. To achieve this, the field `pit_stop` under `'pathfollowing'` in the problem definition file, is set

to `[100,-40]`, where `100` is just a dummy variable for the pathfollowing parameter λ^*, which will not be reached during the run. `counter` is set to `Inf`, assuring that the run will continue until -40 is reached by p and then will stop and save the path.
In the solver settings, a starting mesh with 101 equidistant mesh points, 3 Gaussian collocation points in-between each two of these mesh points and the relative and absolute tolerances for the non-linear solver set to 10^{-6} and the relative and absolute tolerances for the mesh adaptation set to 10^{-4}. $\theta_{\max}$ was set to 10^{-2} and the other pathfollowing specific parameters were all kept at the values presented in Section 1.2.2. The run finishes after 13 tangent continuation steps, without any mesh adaptation. The result is displayed in Figure 2.12.

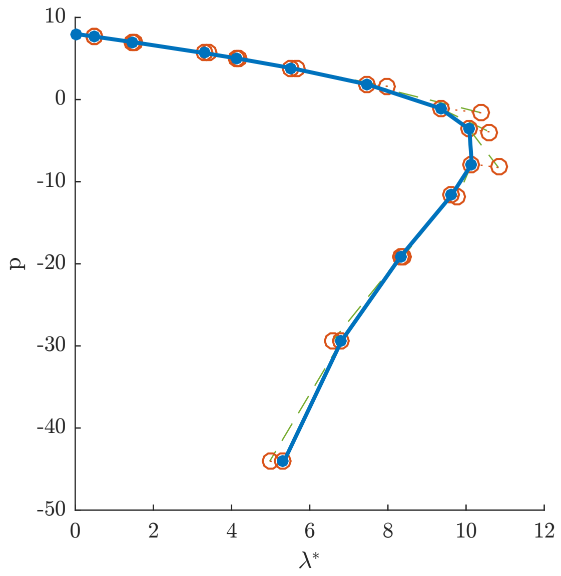


Figure 2.12: 1-dim. Navier-Stokes equation: Evolution of p with $\theta_{\max} = 10^{-2}$ until $p = -40$.

Building on that first run, the `pit_stop` option is set to `[100,-60]` and `startat` to the name of the file, containing the data of the previously saved run in the field `startat`. This is computed in 2 steps, again without any mesh adaptation. The result of this run is displayed in Figure 2.13.
Finally, wanting to see the solution of the problem with λ^* being exactly equal to 0, 5 and 10, the field `require_exact` is set to `[0,5,10]` and the field `only_exact` to `1`. In `p_exact` the results as shown in Figure 2.14, are saved.
Finally, changing the monitored values of the solution from simply the evolution of p to observing the evolution of the characteristic values of the solution $z_1(\frac{1}{2})$, p and $\|z_1\|_\infty$. This is undertaken by creating a new function

```
function ret = PathCharData2(x1,coeff,ordnung,rho)
help = coeffToValues( coeff , x1 , ordnung , rho , 0:1/1000:1 , 0 );
```

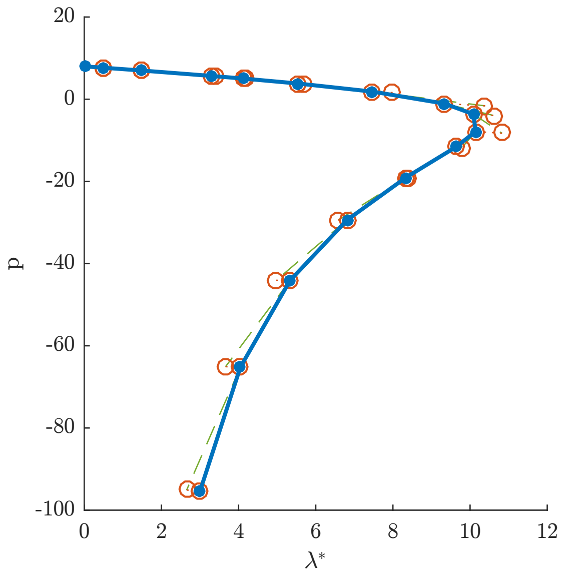


Figure 2.13: 1-dim. Navier-Stokes equation: Evolution of p with $\theta_{\max} = 10^{-2}$ until $p = -60$.

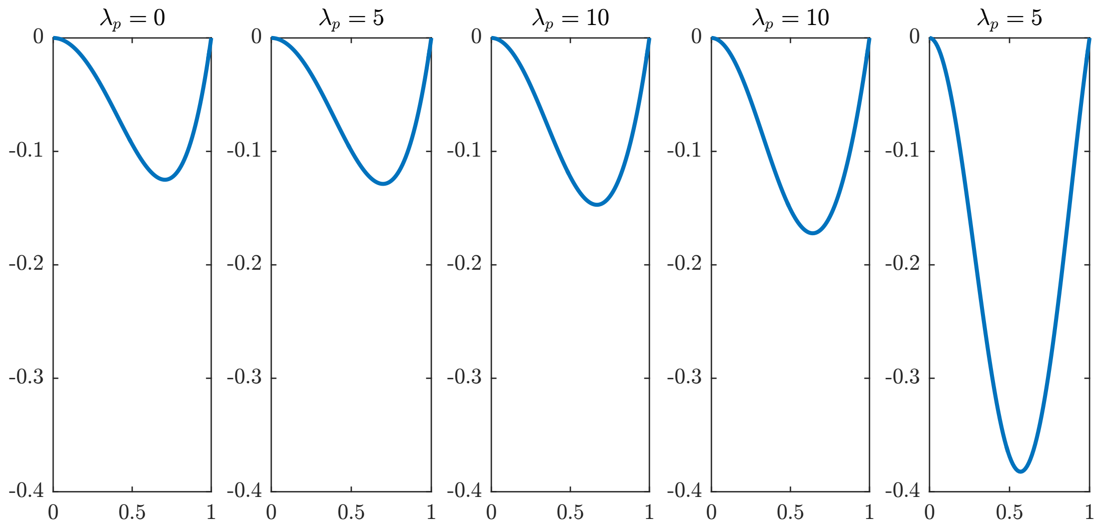


Figure 2.14: 1-dim. Navier-Stokes equation: Approximations to the solutions for $\lambda^* = 0, 5, 10$ along the path from Figure 2.13.

```
ret= [ help(501); coeff(end-1); max(abs(help)) ];
end
```

This can be transmitted to the program as a function handle through the 4th argument of the function bvpsuite2, as is explained in the manual of bvpsuite2.0 [16]. The three resulting paths from this computation are displayed in Figure 2.15.

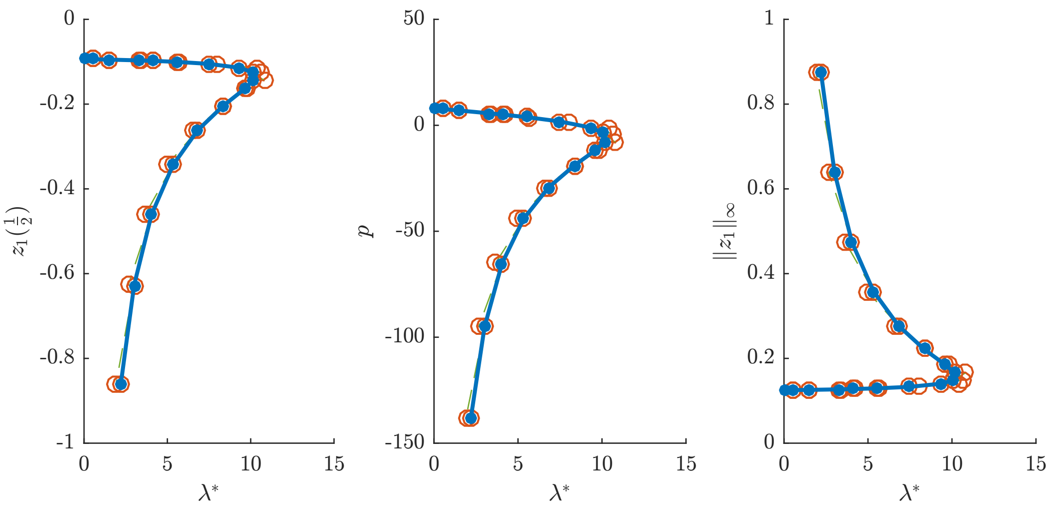


Figure 2.15: 1-dim. Navier-Stokes equation: Evolution of the characteristic values $z_1(\frac{1}{2})$, p and $\|z_1\|_\infty$ respectively under variation of λ^*.

Chapter 3

Further simulations

The simulations in the previous chapters were carried out in an effort to reproduce results, that were already computed in the context of pathfollowing. In this chapter, the focus is set on simulations, which were not computed yet. These problems were considered throughout the length of the creation of this work.

3.1 Variable coefficient Helmholtz Equation

This section will serve as a report on the work which has been done in collaboration with Sukjung Hwang and Sungjin Lee, from the group of Professor Seick Kim of the department of Mathematics at Yonsei University in South Korea. An object of their research was the variable coefficient Helmholtz partial differential equation written as

$$\nabla \big(a(x)\nabla u(x)\big) + k^2 u(x) = 0, \tag{3.1}$$

where $k \in \mathbb{R}$ is a constant and the scalar function a differentiates this equation from the standard Helmholtz equation. The declared goal of their research was the discovery of a fundamental solution for any scalar function a in the equation (3.1). In order to get a sense of what shape this fundamental solution may have, we were asked to provide some simulations. The contact with the research group was established at a time well before the implementation of the pathfollowing module was under way. In the following pages, the results, which were sent to South Korea, are reiterated and some more. In this work, the results were computed with the pathfollowing module, which allowed for a much simpler computation.

3.1.1 Preparation

First, the partial differential equation (3.1) was transformed to an ODE, in order to be able to use `bvpsuite2.0` on it. Under the assumption of radial symmetry in all involved functions, the variable $t := |x| \in \mathbb{R}_0^+$ is introduced and the corresponding ordinary differential equation to (3.1) is

$$a(t)z''(t) + \left(\frac{2}{t}a(t) + a'(t)\right) z'(t) + k^2 z(t) = 0, \qquad \text{for } t \in [0, \infty), \tag{3.2}$$

where $z(t) := u(x)\Big|_{|x|=t}$.
As a model equation, it was first suggested to study

$$a(t) := 1 + \lambda^* e^{-t^2}, \tag{3.3}$$

which tends to 1 as either t tends to ∞ or λ^* tends to 0. Thus equation (3.2) becomes

$$\left(1 + \lambda^* e^{-t^2}\right) z''(t) + \left(\frac{2}{t}\left(1 + \lambda^* e^{-t^2}\right) - 2\lambda^* t e^{-t^2}\right) z'(t) + k^2 z(t) = 0. \tag{3.4}$$

In order to solve this equation for different values of λ^*, some boundary conditions are needed. This differential equation has a singularity at the point $t = 0$. Therefore the boundary conditions need to be chosen with care, i.e. corresponding to the well established theory on ODEs with a singularity in [17] and [18].
Consider the equation (3.4) in an environment of the point $t = 0$, where the singularities blow up and the other terms do not exhibit any special behaviour, then the equation has the following form

$$v''(s) + \frac{2}{s}v'(s) + \frac{k^2}{1+\lambda^*}v(s) = 0.$$

This is rewritten with the definition of $f_1(t) := 2$ and $f_2(t) = \frac{k^2t^2}{1+\lambda^*}$ as

$$v''(s) + \frac{f_1(0)}{s}v'(s) + \frac{f_2(0)}{s^2}v(s) = 0.$$

Since $f_2(0) = 0$, this results in

$$v''(s) + \frac{2}{s}v'(s) = 0. \tag{3.5}$$

The solution ansatz used for this equation is

$$v(t) := t^\beta, \qquad \text{where} \qquad v'(t) = \beta t^{\beta-1} \qquad \text{and} \qquad v''(t) = \beta(\beta-1)t^{\beta-2}.$$

When inputting this into the equation (3.5), the resulting equation for β reads

$$\beta(\beta-1)t^{\beta-2} + 2\beta t^{\beta-2} = 0 \quad \Rightarrow \quad \beta(\beta+1) = 0 \quad \Rightarrow \quad \begin{cases} \beta = 0 & \text{or} \\ \beta = -1. \end{cases}$$

Then the solution to the equation (3.5) is of the form

$$v(t) := c_1 + c_2\frac{1}{t} \qquad \text{with} \qquad v'(t) = c_2\left(-\frac{1}{t^2}\right).$$

In order to obtain a solution $v \in C^2([0,\infty))$, c_2 must be zero and thus the boundary conditions

$$v(0) = c_1 \quad \text{and} \quad v'(0) = 0$$

are imposed. This will guarantee a twice continuously differentiable solution. These results for v and its equation (3.5) are then also valid for z and its equation (3.4).
The condition $z'(0) = 0$ is fixed. To find a fitting initial condition for $z(0)$, the equation is simplified by choosing

$$a(t) \equiv a(0) = 1 + \lambda^* e^0 = 1 + \lambda^*$$

and thus equation (3.2) becomes

$$\left(1+\lambda^*\right) \tilde{z}''(t) + \frac{2}{t}(1+\lambda^*)\tilde{z}'(t) + k^2 \tilde{z}(t) = 0. \tag{3.6}$$

The solution in $\mathbb{R}$ for the case $\lambda^* = 0$, i.e. the solution to the Helmhotz equation, is

$$z_H(t) := \frac{\sin(kz)}{kz}, \tag{3.7}$$

for the standard initial conditions $z(0) = 1$ and $z'(0) = 0$. These can be verified using the L'Hôpital rule. For an arbitrary λ^*, the equation (3.6) is solved by

$$\tilde{z}_H(t) := \frac{1}{\sqrt{1+\lambda^*}} \frac{\sin\left(\frac{k}{\sqrt{1+\lambda^*}} t\right)}{t}, \tag{3.8}$$

which is obtained by scaling the solution (3.7) accordingly. The initial conditions satisfied by $\tilde{z}_H$ are

$$\tilde{z}_H(0) = \frac{k}{1+\lambda^*} \quad \text{and} \quad \tilde{z}_H'(0) = 0. \tag{3.9}$$

These shall be the boundary conditions used in the computation of approximation to the solution of (3.4) under variation of λ^*.

3.1.2 Numerical simulation

For initial value problems posed on a semi-infinite interval, the standard method implemented in `bvpsuite2.0`, which is explained in Section 2.1, is not well suited to approximate a solution. Therefore instead of computing on the whole interval $[0,\infty)$, a finite interval $[0,T]$, $T \in \mathbb{R}$, is chosen.
With the pathfollowing module, the solution of (3.4), with boundary conditions (3.9), is approximated for $\lambda^* \in [0, 20]$. This computation is started with a mesh of 101 points, the relative and absolute tolerances of the non-linear solver are set to 10^{-6} and the relative and absolute tolerances of the mesh adaptation are set to 10^{-4}. The results of this computation are displayed in a surface plot in Figure 3.1.

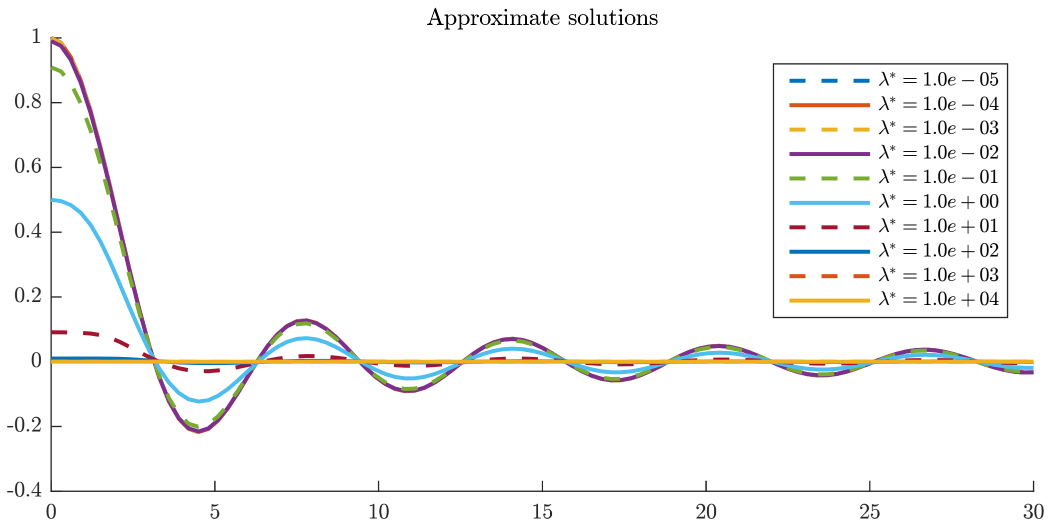


Figure 3.1: Approximations to solutions of (3.4), where $\lambda^* = 10^{-5}, ..., 10^4$.

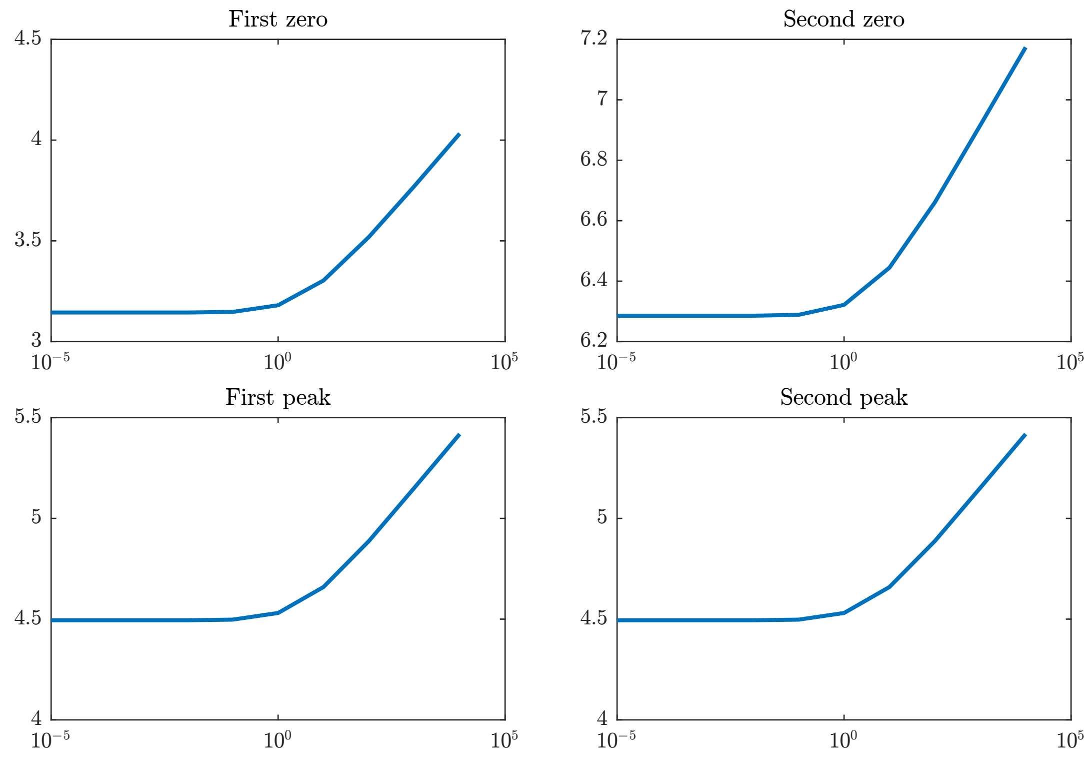


Figure 3.2: Key values of the approximations from Figure 3.1. Position of the 1st zero (upper right), of the 2nd zero (upper left), of the 1st peak (lower right) and of the 2nd peak (lower left), with $\lambda^* = 10^{-5}, ..., 10^4$ in the vertical axis.

The evolution of some characteristic values was also of interest, namely of the position of the first and second zero and the position of the first and second peak. The evolution of these values is displayed in Figure 3.2. In these plots, a shift of these values towards the right is observable, which may hint towards less oscillatory behaviour as λ^* grows. For this problem, the pathfollowing module made it possible to get the results above much faster than it was possible until now. The features which were implemented in the module helped a lot in this quest.
In the following pages, the efforts to find an analytical solution to the equation (3.2) are iterated.

3.1.3 An analytical solution

To search for an analytical solution of (3.2) the Ansatz

$$z_A(t) := \frac{\sin\big(ktf(t)\big)}{kt} g(t)$$

is used, where f and g are unknown scalar functions. When evaluating the equation (3.2) with z_A, then the equation becomes

$$\begin{aligned}
&\Big\{a(t)\Big[\big(2f'(t)+tf''(t)\big)g(t) - \frac{2\big(f(t)+tf'(t)\big)g(t)}{t} + 2\big(f(t)+tf'(t)\big)g'(t)\Big] \\
&\quad + \Big(\frac{2a(t)}{t} + a'(t)\Big)\big(f(t)+tf'(t)\big)g(t)\Big\}\frac{\cos\big(ktf(t)\big)}{t} \\
&\quad + \Big\{a(t)\Big[-k^2\big(f(t)+tf'(t)\big)^2 g(t) + \frac{2g(t)}{t^2} - \frac{2g'(t)}{t} + g''(t)\Big] \\
&\quad + \Big(\frac{2a(t)}{t} + a'(t)\Big)\Big(-\frac{g(t)}{t} + g'(t)\Big) + k^2 g(t)\Big\}\frac{\sin\big(ktf(t)\big)}{kt} \quad = \quad 0
\end{aligned}$$

where the expression has already been ordered in terms with a cosine factor and terms with a sine factor in front. The functions f and g are searched for such that the terms in front of the cosine and the sine respectively are equal to 0.
For the term in front of the cosine, the equation reads

$$\Big[2a(t)\big(f(t)+tf'(t)\big)\Big]\ g'(t) + \Big[\frac{\mathrm{d}}{\mathrm{d}t}\Big(a(t)\big(f(t)+tf'(t)\big)\Big)\Big]\ g(t) \quad = \quad 0.$$

Then the solution g of this equation in dependence of f is

$$g(t) := \frac{1}{\sqrt{a(t)\big(f(t)+tf'(t)\big)}}.$$

For the term in front of the sine, the equation reads

$$a(t)\Big[-k^2\big(f(t)+tf'(t)\big)^2 g(t) + g''(t)\Big] + a'(t)\Big(-\frac{g(t)}{t} + g'(t)\Big) + k^2 g(t) \ = \ 0$$

$$\Leftrightarrow \quad a(t)\frac{g''(t)}{g(t)} + a'(t)\Big(-\frac{1}{t} + \frac{g'(t)}{g(t)}\Big) \;=\; k^2\Big(a(t)\big(f(t)+tf'(t)\big)^2 - 1\Big).$$

By defining $h(t) := a(t)\big(f(t) + tf'(t)\big)$, the equation reads

$$\frac{h''(t)}{2h(t)} - \frac{3}{4}\frac{h'(t)^2}{h(t)^2} + \frac{a'(t)}{a(t)}\Big(\frac{1}{t} + \frac{h'(t)}{2h(t)}\Big) \;=\; \frac{k^2}{a(t)}\Big(1 - \frac{h(t)^2}{a(t)}\Big).$$

A solution was found to this equation, when a is chosen as well. With the pathfollowing parameter λ^*, this is

$$h(t) := \frac{1}{\lambda^* t^2} \quad \text{with} \quad a(t) := \frac{1}{(\lambda^*)^2 t^4} \quad \text{and} \quad k \in \mathbb{R} \quad \Rightarrow \quad f(t) := \frac{\lambda^*}{3}t^3 \quad \text{and} \quad g(t) := t$$
$$\Rightarrow \quad \tilde{z}_1(t) := \frac{1}{k}\sin\Big(\frac{\lambda^*}{3}kt^3\Big) \quad \Rightarrow \quad z_1(t) := \frac{1}{k}\cos\Big(\frac{\lambda^*}{3}kt^3\Big) \tag{3.10}$$

The functions $\tilde{z}_1(t)$ and $z_1(t)$ then solve the equation

$$z''(t) - \frac{2}{t}z'(t) + (\lambda^* k)^2 t^4 z(t) = 0, \qquad \text{for } t \in [0, \infty)\,. \tag{3.11}$$

The function z_1 from (3.10) exhibits a highly oscillatory behaviour, as is shown in Figure 3.3. This makes it very hard to compute this example with `bvpsuite2.0`.

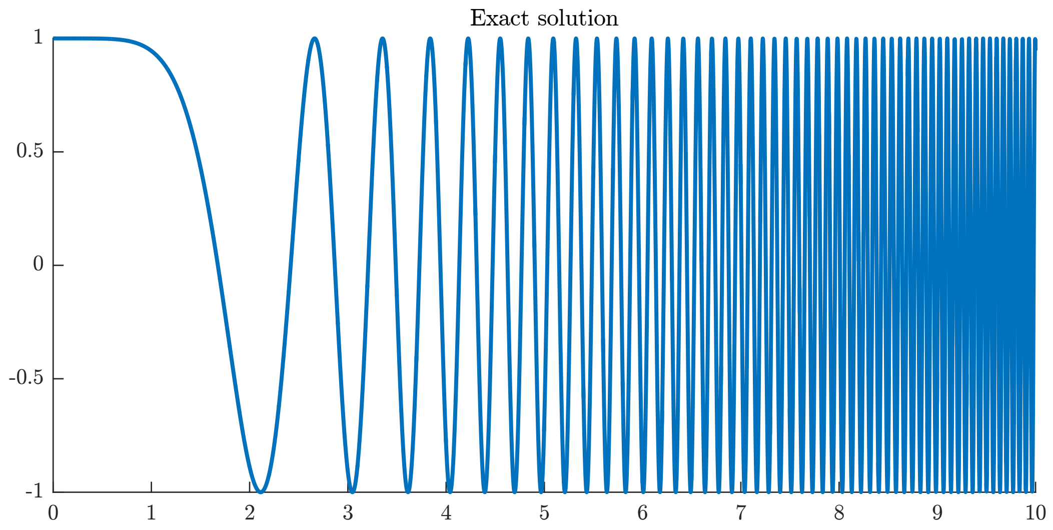


Figure 3.3: Analytical solution (3.10) of (3.11), for $t = [0, 10]$.

This marks the end of the discussion of this problem. The declared goal of finding a fundamental solution to the problem (3.1) was not achieved yet.

Conclusion

At the end of this work, the objective is to reflect on the worth of the results.
The development of the pathfollowing code went through a few stages, where in the first implementation it was based on a Gauss-Newton method, as described in [7]. Soon after, the next implementation was in the form as described in Section 1.2, to fit the already existing code base of `bvpsuite2.0`, especially the non-linear solver routine from `solve_nonlinear_sys`. From then on, the efforts were mainly concentrated on tweaking the code, such that it would compute the problem classes, which is was implemented to deal with, well, and also provide a good user experience. In this process, Katrina Burdeos, visiting student from the University of the Philippines Diliman, was of great help, reporting about and discussing encountered issues and also providing many examples on which to test the code. Many thanks to her for the pleasant and fruitful collaboration.
The aim of the task at hand was clear, but of course not all problems can be solved easily with the implementation at its current state. During the computations for Chapter 3 especially, many obstacles surfaced, to the point that the computations could not be carried out as wished. On the one hand, it was due to the lack of time and computational capacity, which, if more was available, maybe would have helped. On the other hand, for some problems unfortunately it does not seem to be possible to resolve them with the implemented methods, i.e. when the solution components exhibit oscillatory behaviour and a slow decay for problems posed on a semi-infinite interval, as can be seen in Section 3.1.3. Fortunately, the implementation can return some satisfactory results as well, as for the test examples in Section 1.3.
Also, the aim was to make the code as accessible as possible to any user of `bvpsuite2.0`, namely not an expert system. Quite a lot of features have been implemented, which made the code grow quite long. Through comments and shifting recurrent code lines in locally defined functions, it was tried to keep the code in `pathfollowing.m` readable and possibly adaptable for future use. In order to make an easy first use possible to any new user, in the manual [16], a comprehensible usage guide for the pathfollowing example from Section 2.2 was drafted. The jury is still out on whether an accessible code was implemented or not and will probably show itself over time.
For the features that were discussed during the briefing sessions with the supervisors, almost all were added to the code and tested. An idea during development of the code, was that when mesh points were added to the mesh and then a certain condition was triggered, then the number of mesh points would be reduced again for the subsequent steps.

This feature was not tested, since an example, where the solution would get simpler to compute along the path, was not found, therefore neither a reasonable trigger condition. Nevertheless, the corresponding code lines were left in the function `pathfollowing` at the lines `916-934`, just in case such an example would come up and this feature could be properly developed. This goes to show, that there may be more features that would improve the code, but were not considered until now, since they may only be useful in certain special cases.
Finally, it may also be possible to find a more reliable step-length prediction method. In this work, the purpose was to implement the step-length control proposed by Deuflhard et al. ([5], [7] and [9]). The step-length prediction method was then adapted to the case fitting `bvpsuite2.0` of quadratic matrices. In the process, some changes were made to the prediction formula from [5, p. 76, (2.13)], which led to the formula (1.25). During the pathfollowing runs, a lot of times the correction mechanism were triggered, which led to the halving of the step-length and the computation in this step was started over. This correction conditions which are implemented, may be chosen quite strictly by the user, which may lead to more corrections than may be needed to simply follow the path without further requirements. On the other hand, these correction measures revealed themselves to be of importance when computing paths near bifurcation points for instance, as in the shell buckling problem from Section 1.3. If a better step-length control would be needed, the empirical evidence shows the step-length prediction to not work optimally. Refining the prediction formula could lead to better results and less computational effort.

Appendix A

Pathfollowing code

The code that was written for the pathfollowing routine that is described in Chapter 1, is attached in the following pages. It is the file `pathfollowing.m`, which is now part of the MATLAB-package `bvpsuite2.0`.

Listing A.1: `pathfollowing.m`: Pathfollowing routine

```
function [speicher,speicher_exact,tur_pts]=pathfollowing(problem,settings,pathfoll
    ,x1,rho)
% Pathfollowing routine

%% Set options
eval_exact   = 1; % evaluate at the given parameter values
save_ws      = 1; % enable taking step(s) back if desired
dispexact    = 1; % plot at the given parameter values
dispres      = 1; % display the solution and parameter evolution plots
disppredcor  = 1; % display with predictor and corrector step
log          = 1; % log some values
displog      = 1; % plot the logged values
dispmdep     = 1; % plot mesh density evolution
display      = 1; % display messages
% For secondary options, press Ctrl+F and search for 'Secondary'

%% Initializing
m            = length(rho);
xfin         = x1;
counter_mc   = 0; % Counter for mesh correction - DISABLED FOR NOW
ordnung      = feval_problem(problem,'orders');
n            = length(ordnung);
interval     = feval(problem,'interval');
infsplit     = 0;
se_nn        = 40;
% In case the right hand side of the interval is infinite, then x1 is splitted in
    [0,1] and [1,infty) when using splitting interval transformation
selim        = 0;
if interval(2) == Inf
```

```
  if interval(1) == 0 && trafo('splitting')
    infsplit = 1;
    n = n/2;
  end
  se_nn = 40;
  % Secondary option
  % The solution evolution variable 'selim' sets up to which value on the x-axis
     the solutions evolution should be drawn when the solution is computed for a
     problem posed on a semi-infinite interval
  selim = 10;
end
AbsTol     = feval(settings,'absTolSolver');
RelTol     = feval(settings,'relTolSolver');
minmeshpts = max(feval(settings,'minInitialMesh'),length(feval(settings,'mesh')));
     % DISABLED for now

% Set the variables from user input or solver settings
tmp_fn = {'thetaMax','maxCorrSteps','maxSteplengthGrowth','angleMin','
    PredLengthFactor','CorrLengthGrowth','meshFactorMax'};
tmp_val = cell(1,numel(tmp_fn));
for ii=1:numel(tmp_fn)
  if isfield(pathfoll,tmp_fn{ii})
    tmp_val{ii}=pathfoll.(tmp_fn{ii});
  else
    tmp_val{ii}=feval(settings,tmp_fn{ii});
  end
end
[theta_max,maxCorrSteps,maxSteplengthGrowth,cos_min,pred_lf,corr_lf,meshFactorMax]
     = deal(tmp_val{:});

try
  data = pathfoll.pathdata;
catch
  msg = 'There is no ret.pathdata defined in the problem definition. Please refer
     to the manual of bvpsuite2.0 for some examples.';
  error(msg)
end
halve       = 0;
halve_nb    = 0; % times the step-length has been halved
corr_old    = Inf;
min_sl      = 1e-8; % Secondary option: minimal steplength
min_sl_flag = 0; % flag for minimal steplength

psi=zeros(m,max(ordnung)+m,max(ordnung));
for ord=1:max(ordnung)
  for ii=1:m
    psi(ii,1+max(ordnung)-ord:m+max(ordnung),ord)=Psi(ii,rho,ord);
  end
end
```

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

itnum = 0; % Number of completed steps
skip = 0; % Marker in case tagents angle is too steep
% Current number of steps until prompt
cter_flag = 0;
if isfield(pathfoll,'counter')
  counter = pathfoll.counter;
  if counter==Inf || (isfield(pathfoll,'only_counter') && pathfoll.only_counter)
    cter_flag = 1;
  end
else
  counter = 1;
end

prompt='\nType <strong>enter</strong> to carry out the chosen number of steps (ret
    .counter/1 when not modified),\n<strong>p</strong> and <strong>enter</strong>
    to change the maximal length of the predictor step,\n<strong>n</strong> and <
    strong>enter</strong> to change this chosen number,\n<strong>f</strong> and <
    strong>enter</strong> to plot&display solutions of computed steps, and \n<
    strong>s</strong> and <strong>enter</strong> to stop&save the computations\n';
subprompt1='Choose a number of steps: ';
subprompt2='Choose a maximal predictor step length: ';
subprompt3='Choose number of steps to go back: ';
subprompt4='Input a row vector [ . . . ] of steps: ';
subprompt5='Enter one of the options above to proceed: ';
f_size=8; % font size for the plots

% keep the previous interpreter settings and set all to latex
dtiii=get(0, 'DefaultTextInterpreter');
dliii=get(0, 'DefaultLegendInterpreter');
datliii=get(groot, 'DefaultAxesTickLabelInterpreter');
set(0, 'DefaultTextInterpreter', 'latex')
set(0, 'DefaultLegendInterpreter', 'latex')
set(groot, 'DefaultAxesTickLabelInterpreter', 'latex')

%% Prepare what is needed for the options
% load the exact values of the parameter that shall be computed
if isfield(pathfoll,'require_exact')
  exact_val    = sort(unique(pathfoll.require_exact));
  if strcmp(pathfoll.startat,'start')
```

```
        speicher_exact = {};
    end
else
    exact_val = [];
    speicher_exact = [];
end
if isfield(pathfoll,'max_pred_length')
    max_pred = pathfoll.max_pred_length;
    adapt=0;
else
    max_pred = [];
    adapt=0;
end
if isfield(pathfoll,'pit_stop')
    pit_stop = pathfoll.pit_stop;
else
    pit_stop = [];
end
% If only_exact is activated, then only speicher_exact is computed
if isfield(pathfoll,'only_exact') && ~isempty(exact_val)
    only_exact=pathfoll.only_exact;
    if only_exact
        eval_exact=1;
    end
else
    only_exact=0;
end
% to save the workspace in order to go back 'counter'-number of steps in the
    iteration
if save_ws
    prompt_ws='\nDo you want to go back some steps?\nIf yes, press <strong>b</strong>
        and <strong>enter</strong>.\nIf no just <strong>enter</strong>.\n';
    jump_cell = {};
end
% Which plots should be drawn?
if ~eval_exact
    dispexact = 0;
end
if isempty(data) % data should actually never be empty...
    dispres = 0;
end
if ~dispres
    disppredcor = 0;
end
if ~log && displog
    displog = 0;
end
if~(feval(settings,'meshAdaptation'))
    dispmdep = 0;
```

```
end
if dispmdep || dispres || displog
  pathfollplot=figure('units','normalized','outerposition',[0.2 0.2 0.6 0.6],'
     PaperUnits','normalized');
  ax = gca;
  ax.FontSize=f_size;
  clf ;
  pathfollplot.Color = 'White' ;
end
if dispmdep
  % Mesh density evolution plot
  mdep_nn = 50;
  X_mdep  = linspace(0,1,mdep_nn);
  Y_mdep  = [];
  Z_mdep  = [];

  % Mesh evolution plot
  pts_nb = 20;
  if length(x1)<=pts_nb
    filler = interval(2)*ones(1,length(x1)-pts_nb);
    Pts_Mat=[x1 filler];
  else
    pts_step = ceil(length(x1)/pts_nb);
    filler = interval(2)*ones(1,pts_nb-length(x1(1:pts_step:end-1)));
    Pts_Mat = [x1(1:pts_step:end-1) filler];
  end

  % Call the figures
  if dispres
    path_pos = [0 0.5 1 0.5];
    mdep_pos = [0 0 1 0.5];
  else
    mdep_pos = [0 0 1 1];
  end
elseif dispres
  path_pos = [0 0 1 1];
end
if dispres
  figure(pathfollplot);
end
if log
  logstruct.theta_0          = [] ;
  logstruct.theta_max        = [] ;
  logstruct.steplength       = [] ;
  logstruct.steplength_pred = [pathfoll.steplength] ;
  logstruct.sl_adapt         = [];

  logstruct.norm_F           = [];
  logstruct.cpt              = [] ;
  logstruct.orthogonal      = [] ;
  logstruct.max_error       = [];
  logstruct.mesh_length     = [] ;

  logstruct.c_s             = Inf  ;
  logstruct.cos_ab          = Inf;
  logstruct.corr_dist       = Inf ;
  logstruct.norm_delta_0    = [] ;
  logstruct.norm_delta_1    = [] ;

  logstruct.mesha_halved    = [];
  logstruct.angle_halved    = [];
  logstruct.trm_halved      = [];
  logstruct.pdist_halved    = [];
  logstruct.cdist_halved    = [];

  % logstruct fields that are marks in the plot whenever steplength got halved for
     one of the reasons
  % 1. mesh adaptation was too long
  % 2. angle between consecutive steps was too big
  % 3. trust region method was activated
  % 4. corrector step was too long compared to predictor step
  % 5. corrector step was too long compared to previous corrector step
  fn_halve = fieldnames(logstruct);
  fn_halve = fn_halve(end-4:end);
  val_halve = [ 0 1 0 1 10 ];

  if displog
   if dispres && dispmdep
    path_pos = [0.5 0.5 0.5 0.5];
    mdep_pos = [0.5 0 0.5 0.5];
    log_pos  = [0 0 0.5 1];
   elseif dispres
    path_pos = [0.5 0 0.5 1];
    log_pos  = [0 0 0.5 1];
   elseif dispmdep
    mdep_pos = [0.5 0 0.5 1];
    log_pos  = [0 0 0.5 1];
   else
    log_pos  = [0 0 1 1];
   end
  end
else
  logstruct = [];
end
exact_lp = Inf;

%% Preprocess the two cases: Start from initial point or loaded data
% Start at the given initial point on the solution path
```

```
if strcmp(pathfoll.startat,'start')
 % Set the starting value for lambda_p
 lambda_p=pathfoll.start;

 % Before computing a predictor corrector step, the initial solution needs to be
    computed first
 itnum=itnum-1;

 % If the predictor and corrector steps should be drawn
 if disppredcor
  val = {};
 else
  val = [];
 end

 % Inital steplength and initalize struct for predictor, used to carry over
    information from the pathfollowing routine into the bvpsuite2.0 routine
 steplength = pathfoll.steplength;
 predictor  = struct;

 % Save the important values in predictor, that matter for the computation
 predictor.steplength = steplength;
 predictor.infsplit   = infsplit;
 predictor.lpfix      = 0;
 predictor.x1         = x1;
 predictor.lambda_p_0 = lambda_p;
 predictor.skip       = 0;

 % Also some solver settings (needed in PredCorrStrat and meshadaptation.m) and
    display setting
 predictor.maxCorrSteps        = maxCorrSteps;
 predictor.maxSteplengthGrowth = maxSteplengthGrowth;
 predictor.meshFactorMax       = meshFactorMax;
 predictor.display             = display;

 % Starting point data for the Newton iteration in case of nonlinear problem
 a_p=pathfoll.initProfile;

 % Search for the next given parameter value, for which the value should be
    computed, if required
 if ~isempty(exact_val)
  if steplength >0
   exact_lp = find(exact_val>=lambda_p,1);
  else
   exact_lp = find(exact_val<=lambda_p,1,'last');
  end
 end
 % Define some values needing to exist for the function call of Postprocess later,
    they do not have any specific values as of now
 tangent_new=0; theta_0=0; delta_0=0;
else
 try
  filename=strcat(pathfoll.dir,pathfoll.startat);
  load(filename);
 catch MException
  error(MException.message);
 end
 if ~only_exact
  speicher_exact = p_exact;
 else
  speicher_exact = {};
 end
 speicher = p_save(:,1:end-1); % savestruct column is not needed

 % Define x1 and a_p
 sol       = speicher{1,end-1}; % sol is the penultimate solution
 x1        = sol.x1;
 a_0       = [ sol.coeff ; sol.parameters ; sol.lambda_p ] ;
 % Define sol, predictor, xfin and a_c
 sol       = speicher{1,end}; % sol is the last computed solution
 predictor = sol.predictor;
 a_p       = a_0 + predictor.steplength.*predictor.tangent';
 xfin      = sol.x1;
 a_c       = [ sol.coeff ; sol.parameters ; sol.lambda_p ] ;

 savestruct = p_save{1,end};
 if log
  logstruct   = p_save{2,end};
 end

 jump_cell   = p_save{3,end};

 % Reassign all the variables from savestruct
 delta_0         = savestruct.delta_0;
 theta_0         = savestruct.theta_0;
 itnum           = savestruct.itnum;
 val             = savestruct.val;
 exact_lp        = savestruct.exact_lp;
 pred_tmp        = savestruct.pred_tmp;
 corr_old        = savestruct.corr_old;
 if isfield(savestruct,'halve_nb')
  halve_nb    = savestruct.halve_nb;
 else
  halve_nb    = 5;
 end
 if ~exist('pit_stop','var')
```

```
    pit_stop    = savestruct.pit_stop;
end
if dispmdep
  try
    Pts_Mat = savestruct.Pts_Mat;
    Y_mdep  = savestruct.y_mdep;
    Z_mdep  = savestruct.z_mdep;
  catch
    Y_mdep  = [];
  end
end

% If data has been changed, then recompute for the new data
if max(data(xfin,a_c,ordnung,rho)~=speicher{3,end})
  [speicher,speicher_exact,val,corr_old] = correct_data(speicher,speicher_exact,
    val,data,ordnung,rho);
end

steplength = predictor.steplength;
tangent_new = predictor.tangent';
% Compute eval_exact if necessary
if ~isempty(exact_val) && ~only_exact
  eval_exact = 1;
  % if sol.lambda_p<exact_val(1) and step-length<0 or
  tmp_bool1 = sol.lambda_p<exact_val(1) && steplength<0;
  % if sol.lambda_p>exact_val(end) and step-length>0
  tmp_bool2 = exact_val(end)<sol.lambda_p && steplength>0;
  % then for now, eval_exact is set to 0
  if tmp_bool1 || tmp_bool2
    eval_exact=0;
  end
end

% Update solver settings in case it was changed by user when reloading the run
predictor.maxCorrSteps = maxCorrSteps;
predictor.maxSteplengthGrowth = maxSteplengthGrowth;
predictor.meshFactorMax = meshFactorMax;

if ~only_exact
  % Prepare for the next step
  [a_0,a_p,tangent_new,steplength,predictor,delta_0,nda,nsd,theta_0,logstruct,
    eval_exact,exact_lp] = PredCorrStrat(problem,settings,predictor,a_c,a_p,x1,xfin
    ,sol,tangent_new,ordnung,rho,psival,psi,steplength,exact_val,exact_lp,delta_0,
    theta_0,theta_max,log,logstruct,eval_exact,itnum,halve_nb,min_sl);
else
  eval_exact = 1;
  exact_lp = Inf;
end
```

```
    % Accept the new mesh for the next step
    x1 = xfin;

    lambda_p = speicher{2,1};

    itnum=itnum+1;
end
```

```
%% Pathfollowing routine
if itnum==-1 % Initial solution is computed first
    jj = 0;
else % Initial solution was already computed
    jj = 1;
end
% Secondary option:
% After jj_max steps, the routine will stop & save. This number may be changed.
    This is a safety measure for the case where 'counter' is set to Inf.
jj_max = 1e4;
if counter==Inf
    counter=jj_max;
end

% Main loop: is only exited as soon as jj==counter and then 's' is chosen in the
    user prompt, either by the user or automatically.
while 1
    % Prechecks of the step-length
    if itnum~=-1 && ~only_exact
        % Check whether the step-length is shorter than the minimal steplength
        if abs(steplength)<min_sl
            min_sl_flag=1; % Set flag to stop later
            steplength=sign(steplength)*min_sl;
            predictor.steplength=steplength;
            a_p = a_0 + steplength.*tangent_new;
            predictor.lambda_p_p=a_p(end);
            if display
                fprintf('\n  The steplength was shorter than the minimal steplength %1.1e!\n
                 It has been augmented to the minimal steplength.\n',min_sl)
            end
        end
        % Check whether the user-defined maximal step-length is overreached, if it is,
          adapt the predictor to a closer step
        data_tmp=data(x1,a_p,ordnung,rho);
        pred_tmp=sqrt((a_p(end)-speicher{2,end})^2+(max(abs(speicher{3,end}-data_tmp)))
          ^2);
        if ~isempty(max_pred) && pred_tmp>max_pred % pred_tmp~=max_pred
            div_tmp=max_pred/pred_tmp;
            steplength=steplength*div_tmp;
            predictor.steplength=steplength;
            a_p = a_0 + steplength.*tangent_new;
```

```
      predictor.lambda_p_p=a_p(end);

      pred_tmp = max_pred;

      if display
        fprintf('\n  Predictor step was too long. Steplength was reduced to %f.\n', steplength)
      end

      if log
        % save whether the step-length was dimished or not
        adapt = 1;
      end
    end
  end

  if only_exact
    fprintf('\nComputing the approximate solutions for lambda_p=[ require_exact ]:\n')
  elseif itnum~=-1 && display
    fprintf('\n  Starting at predictor value %f:\n\n',a_p(end))
  elseif display
    fprintf('\n  Starting at predictor value %f:\n\n',lambda_p)
  end

  % Corrector Step
  if halve==0
    cpt = cputime;
  end
  if log && itnum~=-1 && ~only_exact
    if skip==0
      for ii=1:length(fn_halve)
        logstruct.(fn_halve{ii}) = [ logstruct.(fn_halve{ii}) val_halve(ii)/(halve==ii) ];
      end
    else
      logstruct.(fn_halve{halve})(end) = val_halve(halve);
    end
  end
  if only_exact
    itnum = itnum-1;
  elseif (feval(settings,'meshAdaptation'))
    [sol,~,halve] = meshadaptation(problem,settings,x1,rho,a_p,predictor);
    % If points have been added in the mesh, then reset the counter, otherwise wait for ? number of iterations in which the mesh does not change, to halve the mesh length if needed
    % ..................... DISABLED FOR NOW .....................
    if length(x1)>length(xfin)
      counter_mc = 0;
    else
      counter_mc = counter_mc + 1;
    end
  elseif(feval(settings,'errorEstimate'))
    if ~isfield(predictor,'tangent') && feval(problem,'linear')
      [~,~,sol]=solveLinearProblem(problem,x1,rho);
    else
      [~,~,sol,halve] = solveNonLinearProblem(problem,settings,x1,rho,a_p,1,predictor
      );
    end
    if itnum==-1
      sol.lambda_p = lambda_p;
    end
    if halve==0
      initCoeff = [sol.coeff', sol.parameters, sol.lambda_p ]';
      sol.initCoeff = initCoeff;
      [sol,~,halve] = errorestimate(sol,problem,settings,predictor);
    end
  else
    if ~isfield(predictor,'tangent_new') && feval(problem,'linear')
      [~,~,sol]=solveLinearProblem(problem,x1,rho);
    else
      [~,~,sol,halve] = solveNonLinearProblem(problem,settings,x1,rho,a_p,1,predictor
      );
    end
    if log
      sol.errest = Inf;
    end
  end
  if itnum==-1 && ~only_exact && isstruct(sol)
    sol.lambda_p = lambda_p;
  end

  % If the number of mesh points does not need halving, then compute the angle
      between the tangents and the lengths of the predictor and corrector steps
  if halve==0 && ~only_exact
    % Save the predictor for possible future use
    sol.predictor=predictor;

    % Save the result of the corrector step on the solution path and the new mesh
      xfin
    a_c = [ sol.coeff ; sol.parameters ; sol.lambda_p ];
    predictor.lambda_p_0 = a_c(end);
    xfin = sol.x1;

    while itnum~=-1
      % Check whether the corrector step is short enough compared to the predictor
      step
      speicher_tmp=speicher;
```

```
speicher_tmp{1,end+1}=sol;
speicher_tmp{2,end}=a_c(end);
speicher_tmp{3,end}=data(xfin,a_c,ordnung,rho);
corr_tmp=sqrt((a_c(end)-a_p(end))^2+(max(abs(speicher_tmp{3,end}-data_tmp)))^2)
;
factor1_tmp=corr_tmp/pred_tmp;
factor2_tmp=corr_tmp/corr_old;
if display
 fprintf('\n  Corrector step is %f-times as long as predictor step.\n',
factor1_tmp)
end
if itnum~=0
 if display
  fprintf('  Corrector step is %f-times as long as the previous corrector step
.\n',factor2_tmp)
 end
end
if factor1_tmp>1/pred_lf
 halve=4;
 if display
  fprintf('\n  Corrector step is at most allowed to be %f-times as long as
predictor step!\n',1/pred_lf)
 end
 if ~min_sl_flag
  break
 end
end
if factor2_tmp>corr_lf
 halve=5;
 if display
  fprintf('\n  Corrector step is at most allowed to be %f-times as long as
previous corrector step!\n',corr_lf)
 end
 if ~min_sl_flag
  break
 end
end

if itnum~=0
 % Compute the cosine between the last computed step and the next tangent
 if (feval(settings,'meshAdaptation')) && (length(x1)~=length(xfin) || max(abs
(x1-xfin))>1e-12)
  initP.initialMesh=xfin;
  initP.parameters = sol.parameters;

  initP.initialValues = coeffToValues(tangent_new, x1,ordnung,rho,xfin);
  initP = initial_coefficients(problem,xfin,initP,rho,0);
  tang_a = [ initP.initialCoeff ; tangent_new(end) ] ;
 else
```

```
                tang_a = tangent_new;
            end
            jac_F = functionFDF( 'DF', problem ,a_c,xfin,psival,psi,rho,[]);
            tang_b = tangente_berechnen(jac_F);
            a_p_b = a_c + sign(steplength)*sign(tang_a.'*tang_b)*tang_b;
            data_tang=data(xfin,a_p_b,ordnung,rho);
            nn_data = length(data_tang);
            a=[(speicher_tmp{2,end}-speicher_tmp{2,end-1})*ones(nn_data,1) speicher_tmp{3,end}-speicher_tmp{3,end-1}];
            b=[(a_p_b(end)-speicher_tmp{2,end})*ones(nn_data,1) data_tang-speicher_tmp{3,end}];
            tmp_ab=zeros(1,nn_data);
            for ii=1:nn_data
                n_a=a(ii,:)*a(ii,:)';
                n_b=b(ii,:)*b(ii,:)';
                tmp_ab(ii)=a(ii,:)*b(ii,:)'/sqrt(n_a*n_b);
            end
            cos_ab=min(tmp_ab);
            if display
                fprintf('  Cosine of the angle between current step and next tangent: %1.1e\n',cos_ab)
            end

        % If the cosine smaller than the user set cosine, then the steplength is halved
            if cos_ab<cos_min
                halve=2;
                if display
                    fprintf('\n  The cosine is smaller than the minimal allowed cosine %1.1e\n',cos_min)
                end
            end
        end
        break
    end
end

if itnum~=-1 && halve~=0 && ~min_sl_flag
    steplength = steplength/2;
    predictor.steplength=steplength;
    a_p = a_0 + steplength.*tangent_new;
    predictor.lambda_p_p=a_p(end);

    if display
        fprintf('  Steplength was halved from %f to %f!\n',2*steplength,steplength)
    end

    itnum=itnum-1; % itnum is augmented by 1 at the end of each loop
    skip = skip+1;
    predictor.skip = skip;
```

```
  counter = counter + 1;
 else % Proceed to finishing the step
  if ~only_exact
   halve_nb = skip;
   halve = 0;
   counter = counter-skip;
   jj = jj-skip;
   skip=0;
   predictor.skip = 0;
   if min_sl_flag && ~isstruct(sol) % pathfollowing is stuck
    msg1='The steplength is reduced too much! The pathfollowing routine has
   trouble following the path with the current solver settings! The current
   mininmal steplength (min_sl) is ';
    msg2='. To change this value look for min_sl at the beginning of the file
   pathfollowing.m in the bvpsuite2.0 package. Otherwise adapting the solver
   settings, especially using mesh adaptation (MA) and choosing the MA tolerances
   only slightly looser than the solver tolerances (i.e. solver tol: 10^-6, MA tol
   : 10^-4), may help.';
    msg=[msg1 num2str(min_sl) msg2];
    error(msg)
   end

   % Save the pathdata of interest in the variable speicher
   if itnum==-1
    speicher{1,1}=sol;
    speicher{2,end}=a_c(end);
    speicher{3,end}=data(xfin,a_c,ordnung,rho);
   else
    speicher=speicher_tmp;
    corr_old=corr_tmp;
   end
  end

  % Check if one of the required values of lambda_p has been passed if yes,
   compute the solution in this point and save the solution in speicher_exact
  if itnum~=-1 && only_exact==1 % only speicher_exact is computed in this run
   % Restart at the beginning & find the next exact value to compute
   lp_a0 = cell2mat(speicher(2,:));
   % Find the value exact_val(exact_lp) that is closest to the starting value of
   the parameter lambda_p
   for ii=2:length(lp_a0)
    tmp_sign = sign(lp_a0(ii)-lp_a0(ii-1));
    if tmp_sign==1
     tmp_line = 1:length(exact_val);
    else
     tmp_line = length(exact_val):-1:1;
    end
    for kk = tmp_line
```

```
          tmp_bool1=tmp_sign*lp_a0(ii-1)<=tmp_sign*exact_val(kk) && tmp_sign*exact_val
          (kk)<tmp_sign*lp_a0(ii);
          % If the required exact value is in-between two values of the path, then
          save ii and kk and break
          if tmp_bool1
            exact_lp=kk; % is the entry number in exact_val
            cter_a0=ii-1; % is the step number in the path leading to a_0
            sol = speicher{1,cter_a0+1}; % is the solution in the path leading to a_c
          exact_val(exact_lp) is in-between a_0(end) and a_c(end)
            break
          end
        end
        if exact_lp~=Inf
          break
        end
      end
      % If the for-loop did not break
      if exact_lp==Inf
        msg='None of the required values in ret.require_exact is within the range of
      the computed path!';
        error(msg)
      end
    elseif itnum~=-1 && ~isempty(exact_val) % Compute lp_a0 and cter_a0 to check
      whether a required value has been passed or not
      lp_a0 = a_0(end);
      cter_a0 = 1;
    end
    % Check whether or not one of the values that should be computed at exactly, was
       already passed, if yes, compute at the value(s) that have been passed and are
      required
    if itnum~=-1 && ~isempty(exact_val) && eval_exact && abs(lp_a0(cter_a0)-sol.
      lambda_p)> abs(lp_a0(cter_a0)-exact_val(exact_lp))
      % More than one value can be in the interval that was computed, thus a while
      statement here
      while eval_exact && abs(lp_a0(cter_a0)-sol.lambda_p)> abs(lp_a0(cter_a0)-
      exact_val(exact_lp))
        if display
          fprintf('\n<strong>Exact solution</strong> at %f computation:\n\n',exact_val
          (exact_lp))
        end

        % If by chance the value of lambda in the last computation was exactly the
      required point, then no computaiton is needed, otherwise it is
        if abs(lp_a0(cter_a0)-exact_val(exact_lp)) > 0
          % Define sol_tmp and a_0_tmp whether only_exact is 1 or 0
          if only_exact
            sol_tmp = speicher{1,cter_a0};
            a_0_tmp = [ sol_tmp.coeff ; sol_tmp.parameters ; sol_tmp.lambda_p ] ;
          else
```

```
                        sol_tmp = sol;
                        a_0_tmp = a_0;
                    end
                    predictor = sol_tmp.predictor;
                    tangent = predictor.tangent';
                    sl_tmp = predictor.steplength;
                    predictor.lpfix = 1;
                    predictor.lambda_p_0 = exact_val(exact_lp);
                    a_p_exact = a_0_tmp + (exact_val(exact_lp)-lp_a0(cter_a0))/tangent(end)*
tangent;
                    if(feval(settings,'meshAdaptation'))
                        [sol_exact] = meshadaptation(problem,settings,x1,rho,a_p_exact,predictor);
                    elseif(feval(settings,'errorEstimate'))
                        [~,~,sol_exact] = solveNonLinearProblem(problem,settings,x1,rho,a_p_exact
,1,predictor);
                        initCoeff = [sol_exact.coeff', sol_exact.parameters, sol_exact.lambda_p ]';
                        sol_exact.initCoeff = initCoeff;
                        sol_exact = errorestimate(sol_exact,problem,settings,predictor);
                    else
                        [~,~,sol_exact] = solveNonLinearProblem(problem,settings,x1,rho,a_p_exact
,1,predictor);
                    end
                    predictor.lpfix = 0;
                else % Solution was already computed
                    if ~only_exact
                        sol_exact = speicher{1,end-1};
                    else
                        sol_exact = speicher{1,cter_a0};
                    end
                    % Load the steplength of the last computed step
                    predictor = sol.predictor;
                    sl_tmp = predictor.steplength;
                end
                speicher_exact{1,end+1} = sol_exact;
                tmp_vct = [ sol_exact.coeff ; sol_exact.parameters ; sol_exact.lambda_p ] ;
                speicher_exact{2,end} = tmp_vct(end);
                speicher_exact{3,end} = data(sol_exact.x1,tmp_vct,ordnung,rho);

                if display
                    fprintf('\n<strong>Exact solution</strong> computed.\n')
                end
                predictor.lambda_p_0 = a_c(end); % In case it was changed

                if dispexact
                    plot_step(sol_exact,f_size,infsplit,n,selim)
                end

                % take the next value of the required values for next time
                if ~only_exact
```

```
      exact_lp = exact_lp + sign(sl_tmp);
      if exact_lp==0 || exact_lp>length(exact_val)
       exact_lp = exact_lp - sign(sl_tmp);
       eval_exact=0;
      end
     else
      % exact_lp = exact_lp+1?
      tmp_bool1= sl_tmp>0 && exact_lp<length(exact_val) && (abs(lp_a0(cter_a0)-sol
    .lambda_p) > abs(lp_a0(cter_a0)-exact_val(exact_lp+1)));
      if tmp_bool1
       exact_lp = exact_lp+1;
      end
      % or exact_lp = exact_lp-1?
      tmp_bool2= sl_tmp<0 && exact_lp>1 && (abs(lp_a0(cter_a0)-sol.lambda_p) > abs
    (lp_a0(cter_a0)-exact_val(exact_lp-1)));
      if tmp_bool2
       exact_lp = exact_lp-1;
      end

    % if neither one, then search for the next exact_val(exact_lp) along the path
      if ~(tmp_bool1 || tmp_bool2)
       exact_lp = Inf;
       for ii=cter_a0+2:length(lp_a0)
        for kk=1:length(exact_val)
         tmp_bool1=exact_val(kk)<lp_a0(ii) && exact_val(kk)>lp_a0(ii-1);
         tmp_bool2=exact_val(kk)>lp_a0(ii) && exact_val(kk)<lp_a0(ii-1);
         % If the required exact value is in-between two values of the path, then
     save ii and kk and break
         if tmp_bool1 || tmp_bool2
          exact_lp=kk;
          cter_a0=ii-1;
          sol = speicher{1,cter_a0+1};
          break
         end
        end
        if exact_lp~=Inf
         break
        end
       end
       if exact_lp==Inf
        if display
         fprintf('\nAll the required exact solutions were computed and saved in
    p_exact.\n')
        end
        % Compute the next required value from here, which is not used in this run
    , since only_exact would lead straight to save&return
        if speicher{1,end}.predictor.steplength >0
         exact_lp = find(exact_val>=sol.lambda_p,1);
         if isempty(exact_lp)
```

```
              exact_val=0;
              exact_lp=length(exact_val);
            end
          else
            exact_lp = find(exact_val<=sol.lambda_p,1,'last');
            if isempty(exact_lp)
              exact_val=0;
              exact_lp=1;
            end
          end
          break
        end
      end
    end
  end
end

% When only_exact is activated, then stop here
if only_exact==1
  jj=counter;
else
  % Compute the predictor and corrector steps, if they are to be plotted
  if itnum~=-1 && isa(val,'cell')
    val{1,end+1} = a_p(end);
    val{2,end} = data(x1,a_p,ordnung,rho);
  end

  if log
    err = max(max(abs(sol.errest)));
    logstruct.max_error = [ logstruct.max_error err];
    logstruct.mesh_length = [logstruct.mesh_length [length(x1) ; length(xfin)]];
    logstruct.cpt = [logstruct.cpt cputime-cpt ] ;

    if itnum~=-1
      logstruct.theta_0 = [ logstruct.theta_0 theta_0 ] ;
      logstruct.theta_max = [ logstruct.theta_max theta_max ];
      logstruct.steplength = [ logstruct.steplength steplength] ;

      logstruct.norm_delta_0 = [logstruct.norm_delta_0 nda ] ;
      logstruct.norm_delta_1 = [logstruct.norm_delta_1 nsd ] ;

      if adapt
        logstruct.sl_adapt = [ logstruct.sl_adapt 0];
      else
        logstruct.sl_adapt = [ logstruct.sl_adapt Inf];
      end

      if length(a_p)~=length(a_c) || max(abs(xfin-x1))>1e-12 % mesh got adapted
        p_ma.lpfix = 1;
        F_tmp1 = functionFDF( 'F', problem ,a_p ,x1,psival,psi,rho,p_ma) ;
        F_tmp2 = functionFDF( 'F', problem ,a_c ,xfin,psival,psi,rho,p_ma) ;
      else
        F_tmp1 = functionFDF( 'F', problem ,a_p ,x1,psival,psi,rho,predictor) ;
        F_tmp2 = functionFDF( 'F', problem ,a_c ,xfin,psival,psi,rho,predictor) ;
      end
      logstruct.norm_F = [logstruct.norm_F [ norm(F_tmp1) ; norm(F_tmp2) ] ] ;
      logstruct.orthogonal = [logstruct.orthogonal [abs(F_tmp1(end)) ; abs(F_tmp2(
  end)) ] ] ;

      if itnum~=0
        logstruct.cos_ab = [logstruct.cos_ab cos_ab];
      end
    end
  end

  % Compute the data for the plot of the mesh density evolution plot
  if dispmdep && itnum~=-1
    x_scaled = (xfin-xfin(1))/(xfin(end)-xfin(1)); % Scaling
    dens =1./(diff(x_scaled)*(length(x_scaled)-1)); % Mesh density
    x_int = x_scaled(1:end-1)+diff(x_scaled)/2;
    new_dens = pchip(x_int,dens,X_mdep);
    Y_mdep = [Y_mdep, itnum+1];
    Z_mdep = [Z_mdep; new_dens ];

    % Mesh evolution
    if length(xfin)<=pts_nb
      filler = interval(2)*ones(1,length(xfin)-pts_nb);
      Pts_Mat=[Pts_Mat; xfin filler];
    else
      pts_step = ceil(length(xfin)/pts_nb);
      filler = interval(2)*ones(1,pts_nb-length(xfin(1:pts_step:end-1)));
      Pts_Mat = [Pts_Mat; xfin(1:pts_step:end-1) filler];
    end
  end

  % If the number of mesh points has been changed from the original number, then
  divide the number of mesh points by 2 if possible, otherwise reduce or keep the
    number of mesh points to minmeshpts
  % ..................... DISABLED FOR NOW .....................
  if counter_mc == 10^10 % Secondary option: Set a lower number
    if length(xfin)/2 > minmeshpts && mod(length(xfin),2)==1
      xfin = xfin(1:2:end);
    elseif length(xfin)/2 > minmeshpts && mod(length(xfin),2)==0
      xfin = [xfin(1:2:end),xfin(end)];
    elseif length(xfin)>minmeshpts
      help = xfin(1:floor(length(xfin)/minmeshpts):end);
      if help(end) == xfin(end)
        xfin = help;
```

```
          else
            xfin = [help xfin(end)];
          end
        end
        counter_mc = 0;
      end

      if halve_nb~=0
        fprintf('\nThe step-length was halved %i times in this step.', halve_nb)
      end

      if itnum==-1
        fprintf('\n<strong>Initial solution computed</strong> for value %f.\n', a_c(end) )
      elseif isempty(max_pred)
        fprintf('\n<strong>Step %i complete</strong> for value %f. (Step %i/%i)\n', itnum+1, a_c(end), jj, counter )
      else
        fprintf('\n<strong>Step %i complete</strong> for value %f with pl=%1.3f and max_pl=%1.3f. (Step %i/%i)\n', itnum+1, a_c(end), pred_tmp, max_pred, jj, counter )
      end

      % If min_sl was reached, then the user can choose whether to continue or not
      if min_sl_flag
        min_sl_flag = 0;
        halve = 0;
        jj=counter;
        if display
          fprintf('\nThe minimal steplength %1.1e was reached!\nChoose whether to continue or stop here.\n',min_sl)
        end
      end

      % Check whether the pit_stop values have been reached
      if itnum~=-1 && ~isempty(pit_stop)
        tmp_bool1= a_0(end)<pit_stop(1) & pit_stop(1)<=a_c(end);
        tmp_bool2= a_0(end)>pit_stop(1) & pit_stop(1)>=a_c(end);
        if tmp_bool1 || tmp_bool2
          jj=counter;
          if display
            fprintf('\nParameter value %f reached!!\nTime for a <strong>pit stop</strong>.\n',pit_stop(1))
          end
          pit_stop(1)=Inf;
        end
        tmp_bool1= speicher{3,end-1}<pit_stop(2) & pit_stop(2)<=speicher{3,end};
        tmp_bool2= speicher{3,end-1}>pit_stop(2) & pit_stop(2)>=speicher{3,end};
        if max(tmp_bool1) || max(tmp_bool2)
```

```
            jj=counter;
            if display
             fprintf('\nData value %f reached!!\nTime for a <strong>pit stop</strong>.\n
    ',pit_stop(2))
            end
            pit_stop(2)=Inf;
          end
        end

        % Save the values to jump back to this state from a later step
        if itnum>0
          savestruct = struct;

          savestruct.delta_0     = delta_0;
          savestruct.theta_0     = theta_0;
          savestruct.itnum       = itnum;
          savestruct.val         = val;
          savestruct.exact_lp    = exact_lp;
          savestruct.pred_tmp    = pred_tmp;
          savestruct.corr_old    = corr_old;
          savestruct.pit_stop    = pit_stop;
          savestruct.halve_nb    = halve_nb;
          if dispmdep
            savestruct.Pts_Mat = Pts_Mat;
            savestruct.y_mdep  = Y_mdep;
            savestruct.z_mdep  = Z_mdep;
          end

          jump_cell{end+1}=savestruct;
        end
      end
    end

    % when the number of steps has been carried out
    if jj==counter
      jj=0;

      % Call the plot
      clf(pathfollplot) ;
      pathfollplot.Color = 'White' ;

      % Draw the path
      if dispres
        % Values of lambda_p
        X_se = cell2mat(speicher(2,:));

        % Mesh points
        if infsplit % Mesh was transformed from [0,infty) to [0,1]
          % In [0,1], only plot every 'DD'th value. This number is a divisor of se_nn.
        KK = 1:30;
        DD = KK(rem(se_nn,KK)==0);
        DD = min(DD(DD>2));
        Y_se = [ linspace(0,1,se_nn/DD+1), fliplr(1./linspace(1/se_nn,1-1/se_nn,se_nn-1)) ];
        selimpt = find(Y_se >= selim,1);
        Y_se = Y_se(1:selimpt);
      else
        Y_se = linspace(interval(1),interval(2),se_nn);
      end

      % Function values at X_se and Y_se
      Z_se = cell(1,n);
      for tt=1:n
        Z_se{tt} = zeros(length(X_se),length(Y_se));
        if infsplit
          for kk=1:itnum+2
            x1tmp = speicher{1,kk}.x1;
            coefftmp = speicher{1,kk}.coeff;
            help = coeffToValues( coefftmp, x1tmp,ordnung,rho,linspace(0,1,se_nn+1));
            help1 = help(tt,1:DD:end); % values on [0,1]
            help2 = help(n+tt,end-1:-1:size(help,2)+se_nn/DD+1-selimpt); % values on (1,selim]
            Z_se{tt}(kk,:)=[help1,help2];
          end
        else
          for kk=1:itnum+2
            help = coeffToValues( speicher{1,kk}.coeff, speicher{1,kk}.x1,ordnung,rho, Y_se);
            Z_se{tt}(kk,:) = help(tt,:);
          end
        end

      end

      % Draw the plots
      if disppredcor && itnum~=-1

        figdataplot(itnum+2,speicher,X_se,Y_se,Z_se,pathfollplot,path_pos,val,problem)
      elseif dispres && itnum~=-1
        figdataplot(itnum+2,speicher,X_se,Y_se,Z_se,pathfollplot,path_pos,zeros(length(speicher{3,1}),1),problem)
      end
    end

    if log && itnum~=-1 && displog % Draw log plot
      if only_exact
        p_ma.lpfix = 1;
        F_tmp2 = functionFDF( 'F', problem ,a_c ,xfin,psival,psi,rho,p_ma) ;
```

```
            end
            figlogdataplot(itnum+1,logstruct,pathfollplot,log_pos,AbsTol+max(F_tmp2)*RelTol)
        end

        if dispmdep && size(Z_mdep,1)>1 % Draw mesh and mesh density evolution plot
            figure(pathfollplot);

            % Mesh density evolution plot
            axes( 'Position', [ mdep_pos(1)+mdep_pos(3)/10, mdep_pos(2)+mdep_pos(4)/10, 5*mdep_pos(3)/10, 7*mdep_pos(4)/10] ) ;
            surf(X_mdep,Y_mdep,Z_mdep)
            zl = zlim;
            zlim([0 zl(2)])
            title('mesh density')
            ax = gca;
            ax.FontSize=f_size;
            ax.Title.FontWeight='normal';

            % Mesh evolution plot
            axes( 'Position', [ mdep_pos(1)+7*mdep_pos(3)/10, mdep_pos(2)+mdep_pos(4)/10, 2*mdep_pos(3)/10, 7*mdep_pos(4)/10] ) ;
            hold on
            for ii = Y_mdep(1)-1:itnum+1
                % Rescale in case meshadaptation was not used from the start but just from a loaded point
                oo=ii-Y_mdep(1)+1;
                plot(Pts_Mat(oo+1,:),ii,'k.','LineWidth',1.5);
            end
            title('mesh evolution')
            ax = gca;
            ax.FontSize=f_size;
            ax.Title.FontWeight='normal';
        end

        % If enabled, go back a step if wished
        if itnum>1 && save_ws && ~cter_flag && ~only_exact
            x_inp = input(prompt_ws,'s');
            switch x_inp
                case 'b' % Go back a number of steps
                    % Let the user input a number of steps to go back
                    while 1
                        y_inp=input(subprompt3);
                        if y_inp>=0 && y_inp>=0 && y_inp<=length(jump_cell)-2 && rem(y_inp,floor(y_inp))==0
                            back_cter=y_inp;
                            break
                        end
```

```
          fprintf('Input should be a natural number between 0 and %i.\n',length(
    jump_cell)-2)
        end

        % Adjust jump_cell and define savestruct
        jump_cell = jump_cell(1:end-back_cter);
        savestruct = jump_cell{end};

        % Load from savestruct
        delta_0         = savestruct.delta_0;
        theta_0         = savestruct.theta_0;
        itnum           = savestruct.itnum;
        val             = savestruct.val;
        exact_lp        = savestruct.exact_lp;
        pred_tmp        = savestruct.pred_tmp;
        corr_old        = savestruct.corr_old;
        pit_stop        = savestruct.pit_stop;
        if isfield(savestruct,'halve_nb')
          halve_nb    = savestruct.halve_nb;
        else
          halve_nb    = 5;
        end
        if dispmdep
          try
            Pts_Mat = savestruct.Pts_Mat;
            Y_mdep  = savestruct.y_mdep;
            Z_mdep  = savestruct.z_mdep;
          catch
            Y_mdep  = [];
          end
        end

        % Adjust speicher
        speicher = speicher(:,1:end-back_cter);

        % Define x1 and a_p
        sol       = speicher{1,end-1}; % sol is the penultimate solution
        x1        = sol.x1;
        a_0       = [ sol.coeff ; sol.parameters ; sol.lambda_p ] ;
        % Define sol, predictor, xfin and a_c
        sol       = speicher{1,end}; % sol is the last computed solution
        predictor = sol.predictor;
        a_p       = a_0 + predictor.steplength.*predictor.tangent';
        xfin      = sol.x1;
        a_c       = [ sol.coeff ; sol.parameters ; sol.lambda_p ] ;

        % Readjust all the fields in logstruct
        if log && back_cter~=0
          tmp_fn = fieldnames(logstruct);
         for kk=1:numel(tmp_fn)
           logstruct.(tmp_fn{kk})=logstruct.(tmp_fn{kk})(:,1:end-back_cter);
         end
        end

        predictor.maxCorrSteps        = maxCorrSteps;
        predictor.maxSteplengthGrowth = maxSteplengthGrowth;
        predictor.meshFactorMax       = meshFactorMax;

        % Load steplength and tangent_new from predictor
        steplength = predictor.steplength;
        tangent_new = predictor.tangent';
        % Compute eval_exact if necessary
        if ~isempty(exact_val)
         tmp_bool1= exact_lp==1 && exact_val(exact_lp)<sol.lambda_p && steplength>0;
         tmp_bool2= exact_lp==length(exact_val) && exact_val(exact_lp)>sol.lambda_p
   && steplength<0;
         if tmp_bool1 || tmp_bool2
           eval_exact=1;
         else
           eval_exact=0;
         end
        end
      end
    end

    % If only_exact is activated or the counter was set to Inf, then speicher and
      speicher_exact are saved, otherwise it depends on the user input
    if only_exact || cter_flag
      x_inp = 's';
    else
      x_inp = input(prompt,'s');
      while x_inp == 'f'
        while 1
        y_inp=input(subprompt4);
        if ~isempty(y_inp) && min(y_inp>=0) && min(y_inp<=itnum+1) && max(rem(y_inp
    +1,floor(y_inp+1)))==0
         break
        end
        fprintf('Inputs must be natural numbers between 0 and %i!\n',itnum+1)
        end
        for ii=y_inp
        plot_step(speicher{1,ii+1},f_size,infsplit,n,selim)
        end
        x_inp=input(subprompt5,'s');
      end
    end
    switch x_inp
      case 'p' % Change maximal steplength
```

```
        while 1
          y_inp=input(subprompt2);
          if ~isempty(y_inp) && y_inp>0
            break
          end
          fprintf('Input must be a positive number!\n')
        end
        max_pred=y_inp;
      case 'n' % Change number of steps to carry out at once
        while 1
          y_inp=input(subprompt1);
          if ~isempty(y_inp) && y_inp>0 && rem(y_inp,floor(y_inp))==0
            break
          end
          fprintf('Input must be a positive natural number!\n')
        end
        counter=y_inp;
      case 's' % Save & stop
        if ~only_exact
          % Save
          savestruct = struct;

          savestruct.delta_0     = delta_0;
          savestruct.theta_0     = theta_0;
          savestruct.itnum       = itnum;
          savestruct.val         = val;
          savestruct.exact_lp    = exact_lp;
          savestruct.pred_tmp    = pred_tmp;
          savestruct.corr_old    = corr_old;
          savestruct.pit_stop    = pit_stop;
          savestruct.halve_nb    = halve_nb;
          if dispmdep
            savestruct.Pts_Mat = Pts_Mat;
            savestruct.y_mdep  = Y_mdep;
            savestruct.z_mdep  = Z_mdep;
          end
        end

        % savestruct is saved at the end of speicher, when the program is started from this point, it will access the last available data in this way
        speicher{1,end+1} = savestruct;
        if log
          speicher{2,end} = logstruct;
        end
        if save_ws
          speicher{3,end} = jump_cell;
        end

        lp_a0 = cell2mat(speicher(2,1:end-1));
```

```
            % Find the turning points in lp_a0
            tmp1 = find( lp_a0(1:end-1)>lp_a0(2:end) );
            tmp2 = find( lp_a0(1:end-1)<lp_a0(2:end) );
            if isempty(tmp1) || isempty(tmp2)
              tur_pts=[];
            else
              tur_pts=zeros(3,length(lp_a0));
              kk=0;
              for ii=2:length(lp_a0)

           if (max(tmp1==ii) && max(tmp2==ii-1)) ||  (max(tmp2==ii) && max(tmp1==ii-1))
                  kk=kk+1;
                  tur_pts(1,kk)=lp_a0(ii-1);
                  tur_pts(2,kk)=lp_a0(ii);
                  tur_pts(3,kk)=lp_a0(ii+1);
                end
              end
              tur_pts=tur_pts(:,1:kk);
            end

            % Reinstate the previous interpreter settings
            set(0, 'DefaultTextInterpreter', dtiii)
            set(0, 'DefaultLegendInterpreter', dliii)
            set(groot, 'DefaultAxesTickLabelInterpreter', datliii)

            return
        end
      end

      if skip==0 % If the step-length needs to be predicted
        % Prepare for the next step
        [a_0,a_p,tangent_new,steplength,predictor,delta_0,nda,nsd,theta_0,logstruct,
          eval_exact,exact_lp] = PredCorrStrat(problem,settings,predictor,a_c,a_p,x1,xfin
          ,sol,tangent_new,ordnung,rho,psival,psi,steplength,exact_val,exact_lp,delta_0,
          theta_0,theta_max,log,logstruct,eval_exact,itnum,halve_nb,min_sl);

        % Accept the new mesh for the next step
        x1 = xfin;
      end

      itnum = itnum+1;
      jj = jj+1;

    end

    end

    %% Local functions
    function [tangente]=tangente_berechnen(DF)
```

```

help1=zeros(length(DF(:,1)),1);
help1(end)=1;

tangente=DF\help1;

tangente=tangente/sqrt(sum(tangente.*tangente));

end

function Psireturn=Psi(n,rho,nr)
i=length(rho);
prod=1;
for s=1:i
 if (s~=n)
  prod=conv(prod,[1 -rho(s)])/(rho(n)-rho(s));
 end
end
for s=1:(nr)
 prod=polyint(prod);
end
Psireturn=prod;
%tabulars for collocation points
end

function [] = figdataplot(ii,speicher,X_se,Y_se,Z_se,pathfollplot,path_pos,val,
    problem)

blue_rgb = [0 0.4470 0.7410];
red_rgb = [0.8500 0.3250 0.0980];
green_rgb = [0.4660 0.6740 0.1880];

figure(pathfollplot);
if isa(val,'cell')
 val1=cell2mat(val(1,:));
 val2=cell2mat(val(2,:));
else
 val2=val;
end
nn=size(val2,1);
n=length(Z_se);

f_size=8; % font size for plot

% data evolution plots
for jj=1:nn
 axes( 'Position', [ path_pos(1)+2*path_pos(3)/20, path_pos(2)+(1+(nn-jj)*10)*
    path_pos(4)/(nn*10), 7*path_pos(3)/20, 8*path_pos(4)/(nn*10)] ) ;
 hold on
  if ~isempty(val)
   for kk=1:ii-1
    if strcmp(problem,'testK_8')
     plot(1./[speicher{2,kk},val1(kk)],[speicher{jj+2,kk},val2(jj,kk)],'Color',
    green_rgb,'LineStyle','--') ;
     plot(1./[val1(kk),speicher{2,kk+1},],[val2(jj,kk),speicher{jj+2,kk+1}],'Color
     ',red_rgb,'LineStyle',':','Marker','o','MarkerSize',6) ;
    else
     plot([speicher{2,kk},val1(kk)],[speicher{3,kk}(jj),val2(jj,kk)],'Color',
    green_rgb,'LineStyle','--') ;
     plot([val1(kk),speicher{2,kk+1},],[val2(jj,kk),speicher{3,kk+1}(jj)],'Color',
    red_rgb,'LineStyle',':','Marker','o','MarkerSize',6) ;
    end
   end
 end
 if strcmp(problem,'testK_8')
   a=speicher(2,1:ii);c=1./cell2mat(a);
 else
   a=speicher(2,1:ii);c=cell2mat(a);
 end
 b=speicher(3,1:ii);d=cell2mat(b);e=d(jj,:);
 plot(c,e,'Color',blue_rgb,'LineStyle','-','Marker','o','MarkerFaceColor',blue_rgb
    ,'MarkerSize',3,'LineWidth',1.5)
 xlabel('pathfollowing parameter')
 if ii>50
   plot(c(50:50:ii),e(50:50:ii),'xw','MarkerSize',5)
 end
 ax = gca;
 ax.FontSize=f_size;
end

% solution evolution plot
if strcmp(problem,'testM_7')
 xstop=8;
 axes( 'Position', [ path_pos(1)+12*path_pos(3)/20, path_pos(2)+11*log_pos(4)/20,
    7*path_pos(3)/20, 8*path_pos(4)/20] ) ;
 subplot(n*nn,2,2:2:2*3*nn)
 surf( [Y_se fliplr(1./Y_se(xstop:end-1))] ,X_se, [Z_se{1} fliplr(Z_se{3}(:,xstop:
    end-1))] )
 title('real part of the solution')
 xlabel('interval on mesh')
 ylabel('pathfollowing parameter')
 zlabel('function value')
 ax = gca;
 ax.FontSize=f_size;

 axes( 'Position', [ path_pos(1)+12*path_pos(3)/20, path_pos(2)+1*log_pos(4)/20,
    7*path_pos(3)/20, 8*path_pos(4)/20] ) ;
 subplot(n*nn,2,2*3*nn+2:2:2*6*nn)
```

```
  surf( [Y_se fliplr(1./Y_se(xstop:end-1))] ,X_se, [Z_se{2} fliplr(Z_se{4}(:,xstop:
     end-1))] )
  title('real part of the solution')
  xlabel('interval on mesh')
  ylabel('pathfollowing parameter')
  zlabel('function value')
  ax = gca;
  ax.FontSize=f_size;
else
  for kk=1:n
    axes( 'Position', [ path_pos(1)+12*path_pos(3)/20, path_pos(2)+(1+(n-kk)*10)*
      path_pos(4)/(n*10), 7*path_pos(3)/20, 8*path_pos(4)/(n*10)] ) ;
    if strcmp(problem,'testK_8')
      surf(Y_se,1./X_se,Z_se{kk})
    else
      surf(Y_se,X_se,Z_se{kk})
    end
    name = strcat('solution function',num2str(kk));
    title(name)
    xlabel('interval on mesh')
    ylabel('pathfollowing parameter')
    zlabel('function value')
    ax = gca;
    ax.FontSize=f_size;
  end
end

end

function [] = figlogdataplot(ii,logstruct,pathfollplot,log_pos,tol)
nrow = 5;
ncol = 2;
f_size=8;
figure(pathfollplot);

axes( 'Position', [ log_pos(1)+2*log_pos(3)/20, log_pos(2)+42*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
plot(0:ii,logstruct.cpt)
hold on ;
plot(1:ii,logstruct.trm_halved(1:ii),'*r')
limsy=get(gca,'YLim');
set(gca,'Ylim',[0 limsy(2)+1]) ;
title('cputime')
sp(1) = gca;
sp(1).FontSize=f_size;
sp(1).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+12*log_pos(3)/20, log_pos(2)+42*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
```

```
plot(1:ii,logstruct.steplength)
hold on ;
plot(1:ii,logstruct.steplength_pred,'k')
plot(1:ii,zeros(1,ii),'k:')
plot(1:ii,logstruct.sl_adapt(1:ii),'*g')
set(gca,'Ylim',[-5/4*max(abs(logstruct.steplength)) 5/4*max(abs(logstruct.
    steplength))]) ;
legend('$s$','predicted $s$')
title('steplength $s$')
sp(2) = gca;
sp(2).FontSize=f_size;
sp(2).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+2*log_pos(3)/20, log_pos(2)+32*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
plot(1:ii,logstruct.theta_0)
hold on
plot(1:ii,ones(1,ii),'k--')
plot(1:ii,logstruct.theta_max,'r--')
%limsy=get(gca,'YLim');
set(gca,'Ylim',[0 3/2*max(abs(logstruct.theta_0))]);%max(1.1,limsy(2))]) ;
title('$\theta_0$')
sp(3) = gca;
sp(3).FontSize=f_size;
sp(3).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+12*log_pos(3)/20, log_pos(2)+32*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
semilogy(1:ii,logstruct.theta_max./logstruct.theta_0)
title('$\Theta_m / \Theta_0$')
sp(4) = gca;
sp(4).FontSize=f_size;
sp(4).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+2*log_pos(3)/20, log_pos(2)+22*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
plot(0:ii,logstruct.mesh_length(1,:))
hold on
plot(0:ii,logstruct.mesh_length(2,:),'k')
plot(1:ii,logstruct.mesha_halved(1:ii),'*r')
legend('at beginning','at end')
set(gca,'Ylim',[0 3/2*max(abs(logstruct.mesh_length(2,:)))]);
title('\# mesh points in step')
sp(5) = gca;
sp(5).FontSize=f_size;
sp(5).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+12*log_pos(3)/20, log_pos(2)+22*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
```

```
semilogy(1:ii,logstruct.norm_F(1,:))
hold on
semilogy(1:ii,logstruct.norm_F(2,:))
semilogy(1:ii,tol.*ones(1,ii),'k--')
legend('$||F(a_p)||$','$||F(a_c)||$')
title('$||F||$')
sp(6) = gca;
sp(6).FontSize=f_size;
sp(6).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+2*log_pos(3)/20, log_pos(2)+12*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
semilogy(1:ii,logstruct.orthogonal(1,:))
hold on
semilogy(1:ii,logstruct.orthogonal(2,:))
legend('$|F(a_p)(end)|$','$|F(a_c)(end)|$')
title('$|F(end)|$')
sp(7) = gca;
sp(7).FontSize=f_size;
sp(7).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+12*log_pos(3)/20, log_pos(2)+12*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
semilogy(1:ii,logstruct.norm_delta_0,'--')
hold on
semilogy(1:ii,logstruct.norm_delta_1,'--')
semilogy(1:ii,logstruct.corr_dist,'--')
semilogy(1:ii,logstruct.pdist_halved(1:ii),'*r')
semilogy(1:ii,logstruct.cdist_halved(1:ii),'*g')
legend('$||\Delta y_0||$','$||\Delta y_1||$','$||a_p-a_c||$')
title('$||...||$')
sp(8) = gca;
sp(8).FontSize=f_size;
sp(8).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+2*log_pos(3)/20, log_pos(2)+2*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
plot(1:ii,logstruct.c_s)
hold on
plot(1:ii,logstruct.cos_ab,'k')
plot(1:ii,logstruct.angle_halved(1:ii),'*r')
legend('$c_s$','cos($ab$)')
title('$c_s$ and cos($ab$)')
sp(9) = gca;
sp(9).FontSize=f_size;
sp(9).Title.FontWeight='normal';

axes( 'Position', [ log_pos(1)+12*log_pos(3)/20, log_pos(2)+2*log_pos(4)/50, 7*
    log_pos(3)/20, 7*log_pos(4)/50] ) ;
```

```
plot(0:ii,logstruct.max_error)
title('Maximal Error')
sp(10) = gca;
sp(10).FontSize=f_size;
sp(10).Title.FontWeight='normal';

for jj=1:nrow*ncol
 xlim(sp(jj),[0 ii+1])
end

end

function [a_0,a_p,tangent_new,steplength,predictor,delta_0,nda,nsd,theta_0,
    logstruct,eval_exact,exact_lp] = PredCorrStrat(problem,settings,predictor,a_c,
    a_p,x1,xfin,sol,tangent_new,ordnung,rho,psival,psi,steplength,exact_val,
    exact_lp,delta_0,theta_0,theta_max,log,logstruct,eval_exact,itnum,halve_nb,
    min_sl)

% In the case of mesh adaptation enabled, take special care to set the new
    starting point of the predictor step
if itnum~=-1 && (feval(settings,'meshAdaptation'))
 % If the mesh was adapted, then compute the tangent and a_0 for the new number of
     mesh points
 if length(x1)~=length(xfin) || max(abs(x1-xfin))>1e-12
  initP.initialMesh=xfin;
  initP.parameters = sol.parameters;

  initP.initialValues = coeffToValues(tangent_new, x1,ordnung,rho,xfin);
  initP = initial_coefficients(problem,xfin,initP,rho,0);
  tangent_new = [ initP.initialCoeff ; tangent_new(end) ] ;
  tangent_new(1:end-1) = tangent_new(1:end-1)/sqrt(sum(tangent_new.'*tangent_new))
    ;

  initP.initialValues = coeffToValues(a_p, x1,ordnung,rho,xfin);
  initP = initial_coefficients(problem,xfin,initP,rho,0);
  a_p = [ initP.initialCoeff ; a_p(end) ] ;

  a_0 = [ sol.coeff ; sol.parameters ; a_c(end) ] ;

  predictor.x1 = xfin;
 else
  a_0 = a_c;
 end
else
 a_0 = a_c;
end

if itnum~=-1
 tangent_old = tangent_new;
```

```
end

% Compute the tangent at the new point
jac_F = functionFDF( 'DF', problem ,a_0,xfin,psival,psi,rho,[]);
tangent_new = tangente_berechnen(jac_F);

% Check for turning point
if itnum~=-1 && sign(tangent_old.'*tangent_new)<0
  fprintf('\n\n      <strong>***</strong> Found <strong>TURNING POINT</strong> near
     %f! <strong>***</strong> \n\n',a_0(end));
  steplength=-steplength;
  % take the next value of the required values for next time
  if ~isempty(exact_val) && eval_exact==1
    if (exact_lp>1 && steplength<0) || (exact_lp<length(exact_val) && steplength>0)
      exact_lp = exact_lp + sign(steplength);
    else
      eval_exact=0;
    end
  elseif ~isempty(exact_val)
    eval_exact = 1;
  end
end

if itnum~=-1
  % Compute the new steplength
  c_s = abs(tangent_new.'*tangent_old) ;
  corr_dist = norm(a_p-a_0) ;
  tmp=2*norm(delta_0)/(c_s^2*corr_dist) ;
  beta = sqrt((theta_max/theta_0)*tmp);
  beta_max = max(min(1,predictor.maxSteplengthGrowth),predictor.maxSteplengthGrowth
     /(2^(0.5*halve_nb)));
  beta=min(beta,beta_max);
  steplength = beta*steplength;

  if predictor.display
    fprintf('\n  Steplength was %3.1e and is predicted to be %3.1e!\n',predictor.
     steplength,steplength);
  end
else
  c_s = 1;
end

if log && itnum~=-1
  logstruct.c_s = [ logstruct.c_s c_s ] ;

  logstruct.corr_dist = [logstruct.corr_dist corr_dist ] ;

  logstruct.steplength_pred = [logstruct.steplength_pred steplength];
end
```

```

% Update predictor
predictor.a_0 = a_0.';
predictor.tangent = tangent_new.';
predictor.steplength = steplength;
predictor.lambda_p_0 = a_0(end);

correct=0;
countercorr=0;
ncorrsteps=predictor.maxCorrSteps;
theta_0_min=Inf; steplength_min=steplength;
while correct~=1
  if abs(countercorr)==ncorrsteps
    steplength=steplength_min;
    predictor.steplength = steplength;
  end

  % Carry out predictor step
  a_p = a_0 + steplength.*tangent_new;

  % Update the value of the pathfollowing variable
  predictor.lambda_p_p = a_p(end);

  % Compute what will be needed to compute the new steplength after the corrector
     step
  F_a = functionFDF( 'F', problem ,a_p,xfin,psival,psi,rho,predictor);
  jac_F_a = functionFDF( 'DF', problem ,a_p,xfin,psival,psi,rho,predictor);
  delta_a = - jac_F_a\F_a ;
  new_a = a_p + delta_a ;
  F_new = functionFDF( 'F', problem ,new_a ,xfin,psival,psi,rho,predictor);
  simplified_delta = - jac_F_a\F_new ;

  nda = norm(delta_a);
  nsd = norm(simplified_delta);

  theta_0 = nsd / nda ;
  if theta_0<theta_0_min
    steplength_min=steplength;
  end

  delta_0=delta_a;
  beta = sqrt(1/2);%sqrt(theta_max/theta_0)
  predictor.steplength = steplength;

  % No Prediction-correction is needed if steplength is smaller than the minimally
     allowed step-length
  if abs(steplength)<min_sl
    break
  end
```

```
 % Prediction-correction is needed if theta_0 is too big
 if theta_0>theta_max*c_s^2 && abs(countercorr)~=ncorrsteps
  sl_tmp=steplength;
  steplength=beta*steplength;
  if predictor.display && countercorr==0
   fprintf('  <strong>Prediction correction:</strong>\n  Steplength is decreased
    from %1.1e to %1.1e',sl_tmp,steplength);
  elseif predictor.display
   fprintf(' to %1.1e',steplength)
  end
  countercorr=countercorr-1;
  % Procedure to increase steplength if needed -- disabled for now
  % elseif theta_0<theta_max*c0^2/4 && abs(countercorr)~=ncorrsteps
  %  sl_tmp=steplength;
  %  steplength=steplength/beta*0.99;
  %  countercorr=countercorr+1;
  %  fprintf('\n Steplength is increased from %f to %f!\n',sl_tmp,steplength);
 else
  if predictor.display && abs(countercorr)~=ncorrsteps
   fprintf('\n  Correction procedure should converge with steplength %3.1e!...\n',
    steplength);
  elseif predictor.display
   fprintf('\n  The steplength %f is used!...\n',steplength);
  end
  correct=1;
 end
end

end

function [speicher,speicher_exact,val,corr_old] = correct_data(speicher,
    speicher_exact,val,data,ordnung,rho)
sol = speicher{1,1};
a_0 = [ sol.coeff ; sol.parameters ; sol.lambda_p ] ;
x1 = sol.x1;
speicher{3,1} = data(x1,a_0,ordnung,rho);
for ii=2:size(speicher,2)
 predictor = speicher{1,ii}.predictor;
 a_p = a_0 + predictor.steplength.*predictor.tangent';
 if iscell(val)
  val{2,ii-1} = data(x1,a_p,ordnung,rho);
 end
 sol = speicher{1,ii};
 a_0 = [ sol.coeff ; sol.parameters ; sol.lambda_p ] ;
 x1 = sol.x1;
 speicher{3,ii} = data(x1,a_0,ordnung,rho);
end
```

```
data_tmp = data(speicher{1,end-1}.x1,a_p,ordnung,rho);
corr_old = sqrt((a_0(end)-a_p(end))^2+(max(abs(speicher{3,end}-data_tmp)))^2);

for ii=1:size(speicher_exact,2)
 sol = speicher_exact{1,ii};
 a_0 = [ sol.coeff ; sol.parameters ; sol.lambda_p ] ;
 x1 = sol.x1;
 speicher_exact{3,ii} = data(x1,a_0,ordnung,rho);
end

end

function [] = plot_step(sol,f_size,infsplit,n,selim)
exact_plot=figure('units','normalized','outerposition',[0.3 0.2 0.4 0.6],'
    PaperUnits','normalized');
ax = gca;
ax.FontSize=f_size;
clf ;
exact_plot.Color = 'White' ;
if infsplit
 selimpt = find((1./sol.x1)>=selim,1,'last');
end
for ii = 1:n
 axes( 'Position', [ 1/10, (1+(n-ii)*10)/(n*10), 8/10, 8/(n*10)] ) ;
 if infsplit
  plot([sol.x1 fliplr(1./sol.x1(selimpt:end-1))],[sol.valx1(ii,:) fliplr(sol.valx1
    (n+ii,selimpt:end-1))],'LineWidth',1.5)
 else
  plot(sol.x1,sol.valx1(ii,:),'LineWidth',1.5)
 end
 plottitle=['solution ',num2str(ii),' for $\lambda_p=$',num2str(sol.lambda_p)];
 title(plottitle)
 ax = gca;
 ax.FontSize=f_size;
 ax.Title.FontWeight='normal';
end
end
```

Bibliography

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

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

[3] G. Kitzhofer, *Numerical treatment of implicit singular BVPs*, Doctoral Thesis, Vienna Univ. of Technology, Vienna, Austria (2009)

[4] G.Kitzhofer, O. Koch and E. B. Weinmüller, *Pathfollowing for essentially singular boundary value problems with application to the complex Ginzburg-Landau equation* BIT Numerical Mathematics 49, Springer Netherlands, pp. 141–160 (2009)

[5] P. Deuflhard, B. Fiedler, P. Kunkel, *Efficient numerical pathfollowing beyond critical points*, SIAM Journal on Numerical Analysis, Vol. 24, No. 4, pp. 912–927 (1987)

[6] P. Deuflhard, *Numerical analysis in modern scientific computing – An introduction*, 2nd Ed., Springer-Verlag, New York, USA (2003)

[7] P. Deuflhard, *Newton methods for nonlinear problems – Affine invariance and adaptive algorithms*, Springer Series in Computational 35, Springer-Verlag Berlin Heidelberg (2004)

[8] G. Kitzhofer, O. Koch, P. Lima and E. B. Weinmüller, *Efficient numerical solution of the density profile equation in hydrodynamics*, Journal of Scientific Computing 32, Springer US, pp. 411–424 (2007)

[9] G. Bader and P. Kunkel, *Continuation and collocation for parameter dependent boundary value problems*, SIAM J. Sci. Stat. Comput. 10, pp. 72–88 (1989)

[10] M. Fallahpour, S. McKee and E.B. Weinmüller, *Numerical simulation of flow in liquid crystals using* MATLAB *software*, Technical Report ASC 24 (2017)

[11] M. Fallahpour, S. McKee and E. B. Weinmüller, *Numerical Simulation of Flow in Smectic Liquid Crystals*, Appl. Numer. Math. 132, pp. 154–162 (2018)

[12] C. J. Budd, S. Chen and R. D. Russell, *New self-similar solutions of the nonlinear Schrödinger equation with moving mesh computations*, Journal of Computational Physics 152, pp. 756–789 (1999)

[13] P. Plechac and V. Sverak, *On self-similar singular solutions of the complex Ginzburg-Landau equation*, Communications on Pure and Applied Mathematics 54, pp. 1215–1242 (2001)

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

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

[16] M. Fallahpour, O. Koch, A. Sass and E. B. Weinmüller, *Manual for* `bvpsuite2.0` *– a* MATLAB *solver for singular BVPs in ODEs, EVPs and DAEs*, in preparation (2019)

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

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

[19] A. Steindl, *Private communication*, Oct. and Nov. (2019)