Kollokationsverfahren für singulär gestörte Differentialgleichungen zweiter Ordnung

Björn Ludwig

Abschlussarbeit im Studiengang
Bachelor of Science
Mathematik, Informatik (NF)

an der FernUniversität in Hagen
im Juli 2017

Inhaltsverzeichnis

1 Einleitung

Wir beschäftigen uns in dieser Arbeit mit der Lösung singulär gestörter Zwei-Punkt Randwertprobleme mit einer linearen Differentialgleichung zweiter Ordnung. Zur näherungsweisen Bestimmung der Lösung verwenden wir die Kollokationsmethode und geben schließlich basierend auf der BERNSTEINbasis die Koeffizientenmatrix eines linearen Gleichungssystems an. Dessen Lösung repräsentiert die BÉZIER-Punkte einer stückweise polynomialen Funktion in BÉZIER-Darstellung, welche das gegebene Randwertproblem in bestimmten Schranken löst. Die Verwendung der BERNSTEIN-Basis ist dabei unter anderem motiviert durch die optimalen Stabilitätseigenschaften unter Störungen der Koeffizienten [7] und die Möglichkeit zur effizienten, numerisch stabilen Ermittlung der Funktionswerte. Details zur Motivation für die Verwendung von stückweise polynomialen Funktionen in Anwendungen wie der unseren, finden sich in [16] und [10]. Bei der Formulierung des Problems und des allgemeinen Lösungsansatzes stützen wir uns vor allem auf DE BOOR und SCHWARTZ [5] und REINHARDT [15]. Beim Übergang zum singulär gestörten Problem folgen wir vor allem LINSS [11]. Zur Arbeit mit den BERNSTEIN-Polynomen stellen neben dem Kurs 01277 *Einführung in Computergrafik* der FernUniversität in Hagen [17] auch BÖHM et al. [3] in den Abschnitten 6 bis 8, HOSCHEK und LASSER [10] in den Abschnitten 4.1 und 4.2, und PRAUTZSCH et al. [2] Kapitel 2 wichtige Grundlagen zur Verfügung. Die Bestimmung einer Menge bestimmter Kollokationspunkte mit besonders günstigen Eigenschaften ermöglicht uns ein Satz aus dem Kurs 01270 *Numerische Mathematik I* der FernUniversität in Hagen [12], der unter anderem auch in [8] genau so zur Anwendung kommt wie bei uns. Das Verfahren wurde objektorientiert in der Programmiersprache Java unter Verwendung der externen Programmbibliothek `org.apache.commons.math3` implementiert, wobei einige wichtige Programmteile zur Effizienzsteigerung unter anderem durch Übersetzungen der Fortran-Implementierungen aus [4] ersetzt wurden. Die Kernbestandteile des ausführlich kommentierten Quelltextes der Implementierung finden sich im Anhang. Anhand verschiedener Beispiele werden die Konvergenzeigenschaften des Verfahrens experimentell untersucht.

2 Grundlagen

Wir greifen auf viele Inhalte der oben genannten Arbbeiten zurück und führen alle wesentlichen benötigten Ergebnisse hier auf, teils mit ausführlichen Beweisen. Wir orientieren uns durchweg für die BERNSTEIN-Basis vor allem an der Form der Darstellung in [3] und an [4] für stückweise polynomiale Funktionen.

2.1 Bernstein-Basis und Algorithmus von de Casteljau

Wir bedienen uns folgender Inhalte des Kapitels 4.3 aus [17] über die BERNSTEINbasis und BÉZIER-Kurven und ergänzen Aussagen zu den Ableitungen aus [3], sowie Beweise, falls diese in den Quellen fehlen. Ein wesentlicher Bestandteil der Arbeit mit den BERNSTEIN-Polynomen stellt der Binomialkoeffizient dar. Wir greifen auf folgende Definition aus [18] zurück.

Definition 2.1.1 (Binomialkoeffizient)**.** *Seien $n, k \in \mathbb{N}_0$. Der Binomialkoeffizient $\binom{n}{k}$ ist definiert durch $\binom{n}{k} := \frac{n!}{k!(n-k)!}$ für alle $0 \leq k \leq n$ und $\binom{n}{k} := 0$ für $k > n$.*

Folgende Rekursion für den Binomialkoeffizienten kommt wiederholt zum Einsatz.

Proposition 2.1.2. *Es ist*

$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}. \tag{1}$$

Definition 2.1.3 (Bernstein-Polynome)**.** *Über dem Intervall $[s,t] \subset \mathbb{R}, s < t$ lässt sich für $n \in \mathbb{N}$ aus den binomischen Formeln entwickeln*

$$(t-s)^n = \Big((t-x)+(x-s)\Big)^n = \sum_{i=0}^{n} \binom{n}{i} (t-x)^{n-i}(x-s)^i, \quad x \in [s,t].$$

Die durch $(t-s)^n$ dividierten Summanden

$${}^t_s B^n_i(x) := \frac{1}{(t-s)^n}\binom{n}{i}(t-x)^{n-i}(x-s)^i \in \Pi_n, \quad i = 0,\dots,n, \tag{2}$$

heißen Bernstein*-Polynome vom Grad n.*

Zur Konsistenzwahrung und Vereinfachung späterer Betrachtungen definieren wir darüber hinaus

$${}^t_s B^n_i(x) := 0, \quad \text{für } i < 0 \text{ oder } i > n.$$

Aus der Definition lassen sich unmittelbar folgende Aussagen ablesen:

Korollar 2.1.4 (Eigenschaften der Bernstein-Polynome)**.** *Die* Bernstein*-Polynome vom Grad $n \in \mathbb{N}$ haben für $x \in [s,t] \subset \mathbb{R}$ folgende Eigenschaften:*

- ***Partition der* 1*:*** $\sum_{i=0}^{n} {}^t_s B^n_i(x) = 1$.
- ***Positivität:*** ${}^t_s B^n_i(x) \geq 0, \quad i = 0,\dots,n.$
- ***Symmetrie:*** ${}^t_s B^n_i(x) = {}^t_s B^n_{n-i}(t-(x-s)), \quad i = 0,\dots,n.$

Darüber hinaus gilt:

Proposition 2.1.5 (Rekursivität der Bernstein-Polynome)**.** *Seien*

$${}^t_s B^n_i, \quad i = 0,\dots,n,$$

die Bernstein*-Polynome vom Grad $n \in \mathbb{N}$ über dem Intervall $[s,t] \subset \mathbb{R}$, dann gilt für $x \in [s,t]$:*

$$\begin{aligned} {}^t_s B^n_i(x) &= \frac{x-s}{t-s}\,{}^t_s B^{n-1}_{i-1}(x) + \left(1 - \frac{x-s}{t-s}\right){}^t_s B^{n-1}_i(x) \\ &= \frac{x-s}{t-s}\,{}^t_s B^{n-1}_{i-1}(x) + \frac{t-x}{t-s}\,{}^t_s B^{n-1}_i(x). \end{aligned}$$

Beweis. Wir zeigen die Behauptung direkt mittels (2) in Definition 2.1.3 und ein wenig Algebra.

$$\begin{aligned} \frac{x-s}{t-s}\,{}^t_s B^{n-1}_{i-1}(x) + \frac{t-x}{t-s}\,{}^t_s B^{n-1}_i(x) &= \frac{x-s}{t-s}\,\frac{1}{(t-s)^{n-1}}\binom{n-1}{i-1}(t-x)^{n-1-(i-1)}(x-s)^{i-1} \\ &\quad + \frac{t-x}{t-s}\,\frac{1}{(t-s)^{n-1}}\binom{n-1}{i}(t-x)^{n-1-i}(x-s)^i \\ &= \frac{1}{(t-s)^n}\binom{n-1}{i-1}(t-x)^{n-i}(x-s)^i \\ &\quad + \frac{1}{(t-s)^n}\binom{n-1}{i}(t-x)^{n-i}(x-s)^i \\ &= \frac{1}{(t-s)^n}\left(\binom{n-1}{i-1} + \binom{n-1}{i}\right)(t-x)^{n-i}(x-s)^i \\ &= \frac{1}{(t-s)^n}\binom{n}{i}(t-x)^{n-i}(x-s)^i \text{ mit (1)} \\ &= {}^t_s B^n_i(x). \end{aligned}$$

□

Der folgende Satz gibt Aufschluss über die Gestalt der ν-ten Ableitung, $\nu \in \mathbb{N}$, eines BERNSTEIN-
Polynoms vom Grad $n \in \mathbb{N}$, welche als Linearkombination von BERNSTEIN-Polynomen vom Grad $n - \nu$ dargestellt werden kann.

Satz 2.1.6. *Seien für $n, \nu \in \mathbb{N}$, und $I := [s,t] \subset \mathbb{R}, s < t$. Dann gilt für die ν-te Ableitung der* BERNSTEIN*-Polynome vom Grad n über I*

$$\frac{d^\nu}{dx^\nu} {}_s^t B_i^n(x) = \frac{1}{(t-s)^\nu} \frac{n!}{(n-\nu)!} \sum_{j=\max(0,i+\nu-n)}^{\min(i,\nu)} (-1)^{\nu+j} \binom{\nu}{j} {}_s^t B_{i-j}^{n-\nu}(x). \tag{3}$$

Beweis. Wieder greifen wir für den Beweis auf (2) in Definition 2.1.3 und zeigen (3) mittels Induktion nach ν. Dabei übertragen wir im Wesentlichen den Beweis von Satz 3.1 aus [6] auf allgemeine Intervalle und ergänzen den Induktionsanfang aus [3]. Für $\nu = 1$ gilt (3) wegen

$$\begin{aligned}
\frac{d}{dx} {}_s^t B_i^n(x) &= \frac{d}{dx} \frac{1}{(t-s)^n} \binom{n}{i} (t-x)^{n-i}(x-s)^i \\
&= \frac{1}{(t-s)^n} \binom{n}{i} \left(i(x-s)^{i-1}(t-x)^{n-i} - (n-i)(x-s)^i(t-x)^{n-i-1} \right) \\
&= \frac{1}{(t-s)^n} \left(\frac{i \cdot n!}{i!(n-i)!} (x-s)^{i-1}(t-x)^{n-i} - \frac{(n-i)\cdot n!}{i!(n-i)!} (x-s)^i (t-x)^{n-i-1} \right) \\
&= \frac{1}{(t-s)^n} \left(\frac{n \cdot (n-1)!}{(i-1)!(n-i)!} (x-s)^{i-1}(t-x)^{n-i} \right. \\
&\quad \left. - \frac{n\cdot(n-1)!}{i!(n-i-1)!} (x-s)^i (t-x)^{n-i-1} \right) \\
&= \frac{n}{(t-s)^n} \left(\frac{(n-1)!}{(i-1)!(n-1-(i-1))!} (x-s)^{i-1}(t-x)^{n-i} \right. \\
&\quad \left. - \frac{(n-1)!}{i!(n-1-i)!} (x-s)^i (t-x)^{n-i-1} \right) \\
&= \frac{n}{t-s} \left(\frac{1}{(t-s)^{n-1}} \binom{n-1}{i-1} (x-s)^{i-1}(t-x)^{n-1-(i-1)} \right. \\
&\quad \left. - \frac{1}{(t-s)^{n-1}} \binom{n-1}{i} (x-s)^i (t-x)^{n-1-i} \right) \\
&= \frac{n}{t-s} \left({}_s^t B_{i-1}^{n-1}(x) - {}_s^t B_i^{n-1}(x) \right) \\
&= \frac{n}{t-s} \sum_{j=0}^{1} (-1)^{1+j} {}_s^t B_{i-j}^{n-1}(x) \\
&= \frac{n}{t-s} \sum_{j=\max(0,i+1-n)}^{\min(i,1)} (-1)^{1+j} \binom{1}{j} {}_s^t B_{i-j}^{n-1}(x), \quad \text{weil } {}_s^t B_{-1}^{n-1} = {}_s^t B_n^{n-1} = 0.
\end{aligned}$$

Wir nehmen nun an, dass für $l = 1, \ldots, \nu$, die Gleichung (3) erfüllt ist, dann folgt

$$\begin{aligned}\frac{d^{\nu+1}}{dx^{\nu+1}} {}_s^t B_i^n(x) &= \frac{d}{dx}\left(\frac{d^\nu}{dx^\nu} {}_s^t B_i^n(x)\right) \\ &= \frac{d}{dx}\left(\frac{1}{(t-s)^\nu}\,\frac{n!}{(n-\nu)!} \sum_{j=\max(0,i+\nu-n)}^{\min(i,\nu)} (-1)^{\nu+j}\binom{\nu}{j} {}_s^t B_{i-j}^{n-\nu}(x)\right),\end{aligned}$$

mit der Induktionsannahme. Weiter ergibt sich

$$\begin{aligned}&= \frac{1}{(t-s)^\nu}\,\frac{n!}{(n-\nu)!} \sum_{j=\max(0,i+\nu-n)}^{\min(i,\nu)} (-1)^{\nu+j}\binom{\nu}{j} \frac{d}{dx} {}_s^t B_{i-j}^{n-\nu}(x) \\ &= \frac{1}{(t-s)^{\nu+1}}\,\frac{(n-\nu)\cdot n!}{(n-\nu)!} \sum_{j=\max(0,i+\nu-n)}^{\min(i,\nu)} (-1)^{\nu+j}\binom{\nu}{j} \Big({}_s^t B_{i-j-1}^{n-\nu-1}(x) \\ &\quad - {}_s^t B_{i-j}^{n-\nu-1}(x)\Big),\end{aligned}$$

mit dem Induktionsanfang. Ausmultiplizieren und Kürzen liefert

$$\begin{aligned}&= \frac{1}{(t-s)^{\nu+1}}\,\frac{n!}{(n-\nu-1)!} \left(\sum_{j=\max(0,i+\nu-n)}^{\min(i,\nu)} (-1)^{\nu+j}\binom{\nu}{j} {}_s^t B_{i-j-1}^{n-\nu-1}(x)\right. \\ &\quad \left. - \sum_{j=\max(0,i+\nu-n)}^{\min(i,\nu)} (-1)^{\nu+j}\binom{\nu}{j} {}_s^t B_{i-j}^{n-\nu-1}(x)\right).\end{aligned}$$

Wir verschieben in der ersten Summe k um eins, fassen in der zweiten die (-1)-Faktoren zusammen und lassen dort den gegebenenfalls auftretenden 0-Summanden ${}_s^t B_{n-\nu}^{n-\nu-1}$ weg. Damit erhalten wir mit $-1 \equiv 1 \pmod 2 \Leftrightarrow (-1)^{a-1} = (-1)^{a+1}, a \in \mathbb{Z}$

$$\begin{aligned}\frac{d^{\nu+1}}{dx^{\nu+1}} {}_s^t B_i^n(x) &= \frac{1}{(t-s)^{\nu+1}}\,\frac{n!}{(n-\nu-1)!} \left(\sum_{j=\max(1,i+\nu+1-n)}^{\min(i,\nu+1)} (-1)^{\nu+j+1}\binom{\nu}{j-1} {}_s^t B_{i-j}^{n-\nu-1}(x)\right. \\ &\quad \left. + \sum_{j=\max(0,i+\nu+1-n)}^{\min(i,\nu)} (-1)^{\nu+j+1}\binom{\nu}{j} {}_s^t B_{i-j}^{n-\nu-1}(x)\right), \text{ und durch Addition von } 0+0, \\ &= \frac{1}{(t-s)^{\nu+1}}\,\frac{n!}{(n-\nu-1)!} \sum_{j=\max(0,i+\nu+1-n)}^{\min(i,\nu+1)} (-1)^{\nu+j+1} \\ &\quad \left(\binom{\nu}{j-1} + \binom{\nu}{j}\right) {}_s^t B_{i-j}^{n-\nu-1}(x) \\ &= \frac{1}{(t-s)^{\nu+1}}\,\frac{n!}{(n-\nu-1)!} \sum_{j=\max(0,i+\nu+1-n)}^{\min(i,\nu+1)} (-1)^{\nu+j+1}\binom{\nu+1}{j} {}_s^t B_{i-j}^{n-\nu-1}(x) \\ &= \frac{1}{(t-s)^{\nu+1}}\,\frac{n!}{(n-(\nu+1))!} \sum_{j=\max(0,i+(\nu+1)-n)}^{\min(i,\nu+1)} (-1)^{\nu+1+j}\binom{\nu+1}{j} {}_s^t B_{i-j}^{n-(\nu+1)}(x),\end{aligned}$$

und wir erhalten (3). □

Insbesondere folgt damit das folgende Korollar.

Korollar 2.1.7 (Rekursivformeln der ersten und zweiten Ableitung der BERNSTEIN-Polynome)**.** *Seien $I := [s,t] \subset \mathbb{R}, s < t$ und $n \in \mathbb{N}$. Dann gilt für die* BERNSTEIN*-Polynome vom Grad n über I*

$$\frac{d}{dx}\,{}_s^tB_i^n(x) = \frac{n}{t-s}\left({}_s^tB_{i-1}^{n-1} - {}_s^tB_i^{n-1}\right)(x), \quad \textit{und}$$
$$\frac{d^2}{dx^2}\,{}_s^tB_i^n(x) = \frac{n(n-1)}{(t-s)^2}\left({}_s^tB_i^{n-2} - 2\,{}_s^tB_{i-1}^{n-2} + {}_s^tB_{i-2}^{n-2}\right)(x),$$

für $i = 0, \ldots, n$.

Die Darstellung einer Funktion mittels der BERNSTEINbasis ist wie folgt definiert.

Definition 2.1.8 (BÉZIER-Darstellung und Kontrollpolygon)**.** *Die Darstellung einer Funktion*

$$g\colon [s,t] \to \mathbb{R}\colon x \mapsto g(x) := \sum_{i=0}^{n} {}_s^tB_i^n(x) \cdot b_i, \quad b_i \in \mathbb{R},$$

heißt BÉZIER*-Darstellung. Die Punkte $b_i, i = 0, \ldots, n$ heißen* BÉZIER*-Punkte und definieren das* BÉZIER- *oder Kontrollpolygon.*

Die Werte einer Funktion in BÉZIER-Darstellung lassen sich mittels des folgenden Algorithmus rekursiv, numerisch stabil berechnen.

Algorithmus 2.1.9 (DE CASTELJAU)**.** ***Gegeben:*** *Eine Funktion*

$$g\colon [s,t] \to \mathbb{R}\colon x \mapsto g(x) := \sum_{i=0}^{n} {}_s^tB_i^n(x) \cdot b_i, \quad b_i \in \mathbb{R},$$

in BÉZIER*-Darstellung und ein $x \in [s,t] \subset \mathbb{R}, s < t$.*
$b_n^0(x) := b_n$
for $r = 1, \ldots, n$ ***do***
  $b_{n-r}^0(x) := b_{n-r}$
  for $j = 1, \ldots, r$ ***do***
    $b_{n-r}^j(x) := \frac{x-s}{t-s} \cdot b_{n-r+1}^{j-1}(x) + \left(1 - \frac{x-s}{t-s}\right) \cdot b_{n-r}^{j-1}(x)$
  end for
end for
Ergebnis: $g(x) = b_0^n(x)$.

Mit dem folgenden Satz liefert der Algorithmus ebenfalls die Ableitungswerte.

Satz 2.1.10. *Sei $g \in \Pi_n$ für $n \in \mathbb{N}$ ein Polynom in* BÉZIER*-Darstellung mit* BÉZIER*-Punkten $b_i, i = 0, \ldots, n$, dann ergibt sich für $\nu \in \mathbb{N}, \nu \leq n$, die ν-te Ableitung $g^{(\nu)}$ von g aus dem Algorithmus von de Casteljau in der $(n-\nu)$-ten Stufe zu*

$$g^{(\nu)}(x) = \frac{1}{(t-s)^\nu}\frac{n!}{(n-\nu)!}\sum_{k=0}^{\nu}(-1)^{\nu-k}\binom{\nu}{k}b_k^{n-\nu}(x).$$

Das folgende Lemma liefert eine Darstellung der Zwischenergebnisse des Algorithmus 2.1.9 von DE CASTELJAU als Linearkombination der BÉZIER-Punkte einer Funktion in BÉZIER-Darstellung, wie sie zur Berechnung der Ableitungen mittels Satz 2.1.10 benötigt wird.

Lemma 2.1.11. *Seien $n \in \mathbb{N}, I := [s,t] \subset \mathbb{R}, s < t$ und $g\colon I \to \mathbb{R}\colon x \mapsto g(x) := \sum_{i=0}^{n} {}_s^t B_i^n(x) \cdot b_i, b_i \in \mathbb{R}$ eine Funktion in* Bézier*-Darstellung. Dann gilt für $x \in I$*

$$b_k^m(x) = \begin{cases} b_k, & m = 0, \\ \sum_{i=k}^{m+k} \binom{m}{i-k} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k} \left(\frac{x-s}{t-s}\right)^{i-k} b_i, & m = 1, \dots, n, \end{cases}$$

für $k = 0, \dots, n-m$.

Beweis. Den ersten Teil der Aussage lesen wir einfach vom Algorithmus ab. Den zweiten Teil der Aussage zeigen wir durch Induktion nach m. Im Induktionsanfang $m = 1$ gilt

$$\begin{aligned} b_{n-r}^1(x) &= \frac{x-s}{t-s} b_{n-r+1}^{1-1}(x) + \left(1 - \frac{x-s}{t-s}\right) b_{n-r}^{1-1}(x) \\ &= \frac{x-s}{t-s} b_{n-r+1}^0(x) + \left(1 - \frac{x-s}{t-s}\right) b_{n-r}^0(x) \\ &= \frac{x-s}{t-s} b_{k+1} + \left(1 - \frac{x-s}{t-s}\right) b_k, \quad \text{mit } k := n-r, \\ &= \binom{1}{1} \left(1 - \frac{x-s}{t-s}\right)^0 \left(\frac{x-s}{t-s}\right)^1 b_{k+1} + \binom{1}{0} \left(1 - \frac{x-s}{t-s}\right)^1 \left(\frac{x-s}{t-s}\right)^0 b_k \\ &= \sum_{i=k}^{k+1} \binom{1}{i-k} \left(1 - \frac{x-s}{t-s}\right)^{1-i+k} \left(\frac{x-s}{t-s}\right)^{i-k} b_i, \end{aligned}$$

für $r = 1, \dots, n$, also $0 \le k \le n-1$. Wir nehmen an, dass die Aussage für $b_k^l(x), l = 1 \dots, m$ gilt und folgern daraus

$$\begin{aligned} b_k^{m+1}(x) &= \frac{x-s}{t-s} b_{k+1}^m(x) + \left(1 - \frac{x-s}{t-s}\right) b_k^m(x) \\ &= \frac{x-s}{t-s} \sum_{i=k+1}^{m+k+1} \binom{m}{i-k-1} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k+1} \left(\frac{x-s}{t-s}\right)^{i-k-1} b_i \\ &\quad + \left(1 - \frac{x-s}{t-s}\right) \sum_{i=k}^{m+k} \binom{m}{i-k} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k} \left(\frac{x-s}{t-s}\right)^{i-k} b_i \\ &= \sum_{i=k+1}^{m+k+1} \binom{m}{i-k-1} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k+1} \left(\frac{x-s}{t-s}\right)^{i-k} b_i \\ &\quad + \sum_{i=k}^{m+k} \binom{m}{i-k} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k+1} \left(\frac{x-s}{t-s}\right)^{i-k} b_i \\ &= \binom{m}{m+k+1-k-1} \left(1 - \frac{x-s}{t-s}\right)^{m-m-k-1+k+1} \left(\frac{x-s}{t-s}\right)^{m+k+1-k} b_{m+k+1} \\ &\quad + \sum_{i=k+1}^{m+k} \binom{m}{i-k-1} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k+1} \left(\frac{x-s}{t-s}\right)^{i-k} b_i \\ &\quad + \sum_{i=k+1}^{m+k} \binom{m}{i-k} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k+1} \left(\frac{x-s}{t-s}\right)^{i-k} b_i \\ &\quad + \binom{m}{k-k} \left(1 - \frac{x-s}{t-s}\right)^{m-k+k+1} \left(\frac{x-s}{t-s}\right)^{k-k} b_k \\ &= \binom{m}{0} \left(1 - \frac{x-s}{t-s}\right)^0 \left(\frac{x-s}{t-s}\right)^{m+1} b_{m+k+1} \end{aligned}$$

$$
\begin{aligned}
&+ \sum_{i=k+1}^{m+k} \left(\binom{m}{i-k-1} + \binom{m}{i-k} \right) \left(1 - \frac{x-s}{t-s}\right)^{m-i+k+1} \left(\frac{x-s}{t-s}\right)^{i-k} b_i \\
&+ \binom{m}{0} \left(1 - \frac{x-s}{t-s}\right)^{m+1} \left(\frac{x-s}{t-s}\right)^{0} b_k \\
&= \binom{m+1}{0} \left(1 - \frac{x-s}{t-s}\right)^{0} \left(\frac{x-s}{t-s}\right)^{m+1} b_{m+k+1} \\
&+ \sum_{i=k+1}^{m+k} \binom{m+1}{i-k} \left(1 - \frac{x-s}{t-s}\right)^{m-i+k+1} \left(\frac{x-s}{t-s}\right)^{i-k} b_i \\
&+ \binom{m+1}{0} \left(1 - \frac{x-s}{t-s}\right)^{m+1} \left(\frac{x-s}{t-s}\right)^{0} b_k \\
&= \sum_{i=k}^{m+1+k} \binom{m+1}{i-k} \left(1 - \frac{x-s}{t-s}\right)^{m+1-i+k} \left(\frac{x-s}{t-s}\right)^{i-k} b_i.
\end{aligned}
$$

□

Als Folgerung aus Algorithmus 2.1.9 und Satz 2.1.10 erhalten wir das folgende Korollar zu den Funktions- und Ableitungswerten an den Intervallenden.

Korollar 2.1.12. *Sei $g \in \Pi_n$ ein Polynom in* Bézier-*Darstellung mit* Bézier-*Punkten $b_i, i = 0, \ldots, n$ definiert über dem Intervall $[s,t] \subset \mathbb{R}, s < t$, dann gilt*

- $g(s) = b_0, \quad g(t) = b_n$, *und*
- *für die ersten Ableitungen im Anfangs- und Endpunkt gilt*

$$g'(s) = \frac{n}{t-s}(b_1 - b_0), \quad g'(t) = \frac{n}{t-s}(b_n - b_{n-1}).$$

2.2 Stückweise polynomiale Funktionen und Splines

In diesem Abschnitt führen wir wichtige Begriffe ein, die uns den Umgang mit der Funktionenklasse ermöglichen, aus der wir unsere Näherungslösung bestimmen wollen. Wir orientieren uns dabei weitestgehend an [4] und bleiben so nah wie möglich an den dort eingeführten Notationen und Begrifflichkeiten, ohne einerseits Übersichtlichkeit und andererseits Klarheit der wenigen benötigten Teilergebnisse aufzugeben. Wir beginnen mit der Wiederholung des Vektorraums reeller Polynome.

Definition 2.2.1 (Menge aller reellen Polynome vom Grad n)**.** *Sei $n \in \mathbb{N}_0$. Wir definieren*

$$\Pi_n := \left\{ p \colon \mathbb{R} \to \mathbb{R} \colon x \mapsto p(x) := \sum_{i=0}^{n} a_i x^i, \quad a_i \in \mathbb{R}, \quad i = 0, \ldots, n, \quad a_n \neq 0 \right\},$$

und nennen Π_n die Menge aller Polynome vom Grad n.

Für Schlussfolgerungen im Abschnitt 4 benötigen wir den Fundamentalsatz der Algebra.

Satz 2.2.2 (Fundamentalsatz der Algebra)**.** *Ein Polynom p vom Grad $n \geq 0$*

$$p(x) = \sum_{i=0}^{n} a_i x^i, \quad x \in \mathbb{C},$$

besitzt genau n Nullstellen $z_i \in \mathbb{C}, i = 1, \ldots, n$, ihrer Vielfachheit nach gezählt und lässt sich als Produkt von Linearfaktoren darstellen

$$\begin{aligned} p(x) &= a_n(x - z_1) \cdots (x - z_n) \\ &= a_n \prod_{i=1}^{n} (x - z_i). \end{aligned}$$

Mittels der bekannten Definition von reellen Polynomen definieren wir nun stückweise polynomiale Funktionen.

Definition 2.2.3 (Menge aller stückweise polynomialen Funktionen vom Grad n über $\mathbb{R}$)**.** *Seien $n \in \mathbb{N}_0, l \in \mathbb{N}$, und $\xi := \{\xi_0, \ldots, \xi_l\} \subset \mathbb{R}, \xi_0 < \ldots < \xi_l$. Wir definieren*

$$\Pi_{n,\xi} := \left\{ f \colon \mathbb{R} \to \mathbb{R} \colon x \mapsto f(x) := \begin{cases} p_0(x) & \text{, falls } x < \xi_1, \\ p_i(x) & \text{, falls } \xi_i \leq x < \xi_{i+1}, \quad i = 1, \ldots, l-2, \\ p_{l-1}(x) & \text{, falls } x \geq \xi_{l-1}, \end{cases} \right\},$$

mit $p_0, \ldots, p_{l-1} \in \Pi_n$, und nennen $\Pi_{n,\xi}$ die Menge aller durch die Knotenfolge ξ definierten stückweise polynomialen Funktionen vom Grad n über $\mathbb{R}$.

Die Ableitung einer stückweise polynomialen Funktion definieren wir intiutiv wie folgt.

Definition 2.2.4 (Ableitung einer stückweise polynomialen Funktion über $\mathbb{R}$)**.** *Seien $n, \nu \in \mathbb{N}_0, l \in \mathbb{N}, \xi := (\xi_i)_{i=0}^{l}$ eine streng monoton wachsende Folge in $\mathbb{R}, p_0, \ldots, p_{l-1} \in \Pi_n$, und $f \in \Pi_{n,\xi}$ definiert durch $p_0, \ldots, p_{l-1}$ und ξ. Wir definieren für $x \in \mathbb{R}$*

$$f^{(\nu)}(x) := \begin{cases} p_0^{(\nu)}(x) & \text{, falls } x < \xi_1, \\ p_i^{(\nu)}(x) & \text{, falls } \xi_i \leq x < \xi_{i+1}, \quad i = 1, \ldots, l-2, \\ p_{l-1}^{(\nu)}(x) & \text{, falls } x \geq \xi_{l-1}. \end{cases}$$

Stückweise polynomiale Funktionen, die zusätzlich gewissen Stetigkeitsanforderungen auf einem Intervall genügen, fassen wir mittels des folgenden Begriffs noch enger.

Definition 2.2.5 (C^ν-Spline vom Grad n)**.** *Seien $n \in \mathbb{N}_0, l, \nu \in \mathbb{N}$ und $\xi := (\xi_i)_{i=0}^{l}$ eine streng monoton wachsende Folge in $\mathbb{R}$. Wir definieren*

$$\Pi_{n,\xi,\nu} := \{ f \in \Pi_{n,\xi} | f \in C^\nu[\xi_0, \xi_l] \},$$

und nennen $\Pi_{n,\xi,\nu}$ die Menge aller im Intervall $[\xi_0, \xi_l]$ (mindestens) C^ν-stetigen, stückweise polynomialen Funktionen oder kurz die Menge aller C^ν-Splines vom Grad n auf ξ.

2.3 Orthogonale Polynome

In Abschnitt 4 werden wir sehen, dass wir zur Sicherung bestimmter Konvergenzeigenschaften im klassischen Fall an entscheidender Stelle auf eine bestimmte Folge orthogonaler Polynome zurückgreifen sollten und führen deshalb einige Grundlagen zu dem Thema auf, die allesamt aus Kapitel 5 in [12] stammen. Im Wesentlichen geben wir damit den ausführlichen Beweis für den in unserem Fall wichtigen Teil der Folgerung 7.3.4. aus [12], den wir in Korollar 2.3.6 ausformulieren. Den Begriff der Gewichtsfunktion und den Satz über das durch eine Gewichtsfunktion gegebene Skalarprodukt und die induzierte Norm benötigen wir für die zentrale Definition einer Folge orthogonaler Polynome.

Definition 2.3.1 (Gewichtsfunktion). *Sei $I := (a, b) \subset \mathbb{R}$ ein offenes Intervall. Eine Funktion $\varrho\colon I \to \mathbb{R}_+$, für die das Intergral $\int_a^b p(x)\varrho(x)\mathrm{d}x$, für alle $p \in \Pi$ existiert, nennen wir eine Gewichtsfunktion über I.*

Satz 2.3.2 (Skalarprodukt auf Π und induzierte Norm). *Bei gegebener Gewichtsfunktion $\varrho\colon (a, b) \to \mathbb{R}_+$ ist*

$$\langle v, w\rangle_\varrho := \int_a^b v(x)w(x)\varrho(x)\mathrm{d}x$$

ein Saklarprodukt auf Π und

$$\|v\|_\varrho := \sqrt{\langle v, v\rangle_\varrho},$$

die durch dieses Skalarprodukt induzierte Norm.

Definition 2.3.3 (Folge orthogonaler Polynome). *Seien $I := (a, b) \subset \mathbb{R}$ ein offenes Intervall, $\varrho\colon I \to \mathbb{R}$ eine Gewichtsfunktion und $\langle\cdot,\cdot\rangle_\varrho$ das zugehörige Skalarprodukt. Eine Folge $(u_i)_{i=0}^\infty$ von Polynomen $u_i \in \Pi_i, u_i \neq 0, i = 0, 1, 2, \ldots$, mit*

$$\langle u_i, u_j\rangle_\varrho = 0 \quad \textit{für} \quad i \neq j,$$

wird als Folge von (bezüglich ϱ) orthogonalen Polynomen bezeichnet.

Entscheidende Eigenschaften von Folgen orthogonaler Polynome halten wir in der folgenden Bemerkung fest.

Bemerkung 2.3.4. *Die Mengen $\{u_i\}_{i=0}^\infty$ und $\{u_i\}_{i=0}^n, n \in \mathbb{N}$ bilden orthogonale Basen in Π beziehungsweise in Π_n. Es gilt $\operatorname{Grad} u_i = i$ und somit $u_i \in \Pi_i \setminus \Pi_{i-1}$. Ferner sind die u_i orthogonal zu allen Polynomen $q \in \Pi_{i-1}$, das heißt $\langle u_i, q\rangle_\varrho = 0$ für alle $q \in \Pi_{i-1}, i = 0, 1, 2, \ldots$.*

Die schon erwähnte wichtige Folge von orthogonalen Polynomen wird im Folgenden definiert.

Definition 2.3.5 (Legendre-Polynome). *Die bezüglich der Gewichtsfunktion*

$$\varrho_P\colon (-1, 1) \to \mathbb{R}_+ \colon x \mapsto \varrho_P(x) = 1$$

orthogonalen Polynome $P_i \in \Pi_i$ und $P_i(1) = 1, i = 0, 1, \ldots$, werden als Legendre-*Polynome bezeichnet.*

Die Besonderheit dieser Polynome wird in folgendem Korollar formuliert.

Korollar 2.3.6. *Seien $n \in \mathbb{N}$ und $P_n \in \Pi_n$ das n-te* Legendre-*Polynom mit den Nullstellen $\rho_1, \ldots, \rho_n$. Dann gilt für alle $q \in \Pi_{n-1}$*

$$\int_{-1}^1 q(x) \prod_{i=1}^n (x - \rho_i) = 0.$$

Beweis. Nach dem Fundamentalsatz der Algebra 2.2.2 besitzt P_n genau n Nullstellen $\rho_1, \ldots, \rho_n$ und es gilt

$$P_n(x) = a_n \prod_{i=1}^n (x - \rho_i), \quad x \in \mathbb{R}, \tag{4}$$

für ein $a_n \in \mathbb{R}$. Da per Definition $P_n \neq 0, n = 0, 1, 2, \ldots$, ist offenbar $a_n \neq 0$. Aus Bemerkung 2.3.4 folgt für alle $q \in \Pi_{n-1}$

$$\begin{aligned} 0 = \langle q, P_n \rangle_{\varrho_P} &= \int_{-1}^{1} q(x) P_n(x) \cdot 1 \, \mathrm{d}x \\ &= \int_{-1}^{1} q(x) a_n \prod_{i=1}^{n} (x - \rho_i), \quad \text{mit (4)} \\ &= a_n \int_{-1}^{1} q(x) \prod_{i=1}^{n} (x - \rho_i), \end{aligned}$$

und daraus die Behauptung. □

Zur konkreten Bestimmung der Nullstellen verwenden wir den folgenden Satz.

Satz 2.3.7 (Äquivalenz von Nullstellen orthogonaler Polynome und Eigenwerten der zugeordneten Jacobi-Matrix)**.** *Seien $I \subset \mathbb{R}$ ein offenes Intervall, ϱ eine Gewichtsfunktion über I und $(u_i)_{i=0}^{\infty}$ eine Folge bezüglich ϱ orthogonaler Polynome. Dann sind die Nullstellen von $u_n, n \in \mathbb{N}_0$, und die Eigenwerte der zugeordneten* Jacobi*-Matrix*

$$J_{u,n} := \begin{pmatrix} \alpha_0 & \chi_1 & & 0 \\ \chi_1 & \alpha_1 & \ddots & \\ & \ddots & \ddots & \chi_{n-1} \\ 0 & & \chi_{n-1} & \alpha_{n-1} \end{pmatrix}, \quad \chi_i := \frac{\gamma_{i-1}\delta_i}{\gamma_i \delta_{i-1}}, \quad \alpha_i := \frac{\langle \mu_1 u_i, u_i \rangle_{\varrho}}{\delta_i^2},$$

identisch.

Angewendet auf die Legendre-Polynome ergibt sich die zugeordnete Jacobi-Matrix wie in Aufgabe 5.3.31 angegeben.

Bemerkung 2.3.8. *Es ist*

$$J_{P,n} := \begin{pmatrix} 0 & \frac{1}{\sqrt{3}} & & & 0 \\ \frac{1}{\sqrt{3}} & 0 & \frac{2}{\sqrt{15}} & & \\ & \frac{2}{\sqrt{15}} & 0 & \ddots & \\ & & \ddots & \ddots & \frac{n-1}{\sqrt{4(n-1)^2-1}} \\ 0 & & & \frac{n-1}{\sqrt{4(n-1)^2-1}} & 0 \end{pmatrix}$$

die dem n-ten Legendre*-Polynom zugeordnete* Jacobi*-Matrix.*

2.4 Notationen

Wir verwenden folgende nicht allgemein übliche Notationen. Sei $I := [s, t] \subset \mathbb{R}, s < t$, dann definieren wir für den häufig im Algorithmus von de Casteljau auftretenden Quotienten eine Funktion $\mu \colon I \to [0, 1]$ und schreiben

Notation 2.4.1. $\mu := \mu(x) = \frac{x-s}{t-s}, \quad x \in I.$

Die maximale Schrittweite für unser in Abschnitt 3.4 definiertes Gitter bezeichnen wir als

Notation 2.4.2. $h := \max_{i=0,\ldots,l-1}(\xi_{i+1} - \xi_i)$.

Falls nichts anderes angegeben ist, bezeichnen wir als die Maximumnorm einer Funktion f

Notation 2.4.3. $\|f\|_\infty := \max_{x\in[s,t]}|f(x)|$,

mit dem betrachteten Intervall $I := [s,t]$ unseres Randwertproblems, wie wir es in Abschnitt 3 definieren werden. Für den Wert einer Funktion u an der Stelle τ_j schreiben wir

Notation 2.4.4. $u_j := u(\tau_j)$,

mit der Definition der τ_j, wie wir sie in Abschnitt 3.2.1 einführen werden.

3 Die Aufgabe

Wir betrachten das Zwei-Punkt Randwertproblem

$$-\varepsilon y'' - py' + qy = f, \quad x \in I := [s,t] \subset \mathbb{R}, s < t, \quad y(s) = y_s, y(t) = y_t, \quad y_s, y_t \in \mathbb{R}, \quad \varepsilon \in (0,1],$$

mit einer für $\varepsilon \to 0$ singulär gestörten, linearen Differentialgleichung zweiter Ordnung und auf I mindestens stetigen Funktionen $p, q, f \in C(I)$. Wir setzen im Folgenden voraus, dass eine eindeutige Lösung existiert. Im weiteren Verlauf dieser Arbeit bezeichnen wir die Wahl von $\varepsilon = 1$ als den klassischen und $\varepsilon < 1$ als den singulär gestörten Fall.

3.1 Singuläre Störungen

Im Allgemeinen bezeichnen wir eine von einem kleinen, positiven Parameter ε abhängige Differentialgleichung, deren Verhalten sich beim Grenzübergang $\varepsilon \to 0$ grundlegend ändert, als singulär gestört. Zur formalen Definition des Begriffs der singulären Störung greifen wir auf die Einleitung in [11] zurück.

Definition 3.1.1 (Singularität einer Funktion und singuläre Störung eines Problems)**.** *Sei B ein Funktionenraum und $\|\cdot\|_B$ die dazugehörige Norm. Sei $D \subset \mathbb{R}^d$ ein Parameterdefinitionsbereich. Genau dann nennen wir die stetige Funktion $u\colon D \to B\colon \varepsilon \mapsto u(\varepsilon)$ regulär für $\varepsilon \to \varepsilon^* \in \partial D$, wenn es eine Funktion $u^* \in B$ gibt, so dass*

$$\lim_{\varepsilon\to\varepsilon^*} \|u - u^*\|_B = 0.$$

Andernfalls nennen wir u singulär für $\varepsilon \to \varepsilon^*$. Sei weiterhin $(\mathcal{P}_\varepsilon)$ ein Problem mit der Lösung $u(\varepsilon) \in B$ für alle $\varepsilon \in D$. Genau dann nennen wir $(\mathcal{P}_\varepsilon)$ singulär gestört für $\varepsilon \to \varepsilon^* \in \partial D$ in der Norm $\|\cdot\|_B$, wenn u für $\varepsilon \to \varepsilon^*$ singulär ist.

In Abhängigkeit der konkreten Gestalt unserer Differentialgleichung unterscheiden wir zwei Fälle, die wir wie üblich in Anlehnung an die Realweltprozesse bezeichnen, in deren modellhaften Beschreibungen sie auftreten.

1. Konvektionsdiffusionsprobleme:

   $$-\varepsilon y'' - py' + qy = f, \quad p(x) \geq \beta, \quad x \in [s,t],$$

   für ein $\beta > 0$. Bei solchen Gleichungen tritt typischerweise eine Grenzschicht bei $x = s$ auf, deren Verhalten sich durch die Grenzschichtfunktion $e^{\frac{-\beta x}{\varepsilon}}$ beschreiben lässt. Eine detaillierte Analyse des allgemeinen Verhaltens von Problemen dieser Art findet sich in Abschnitt 3.4.1 von [11].

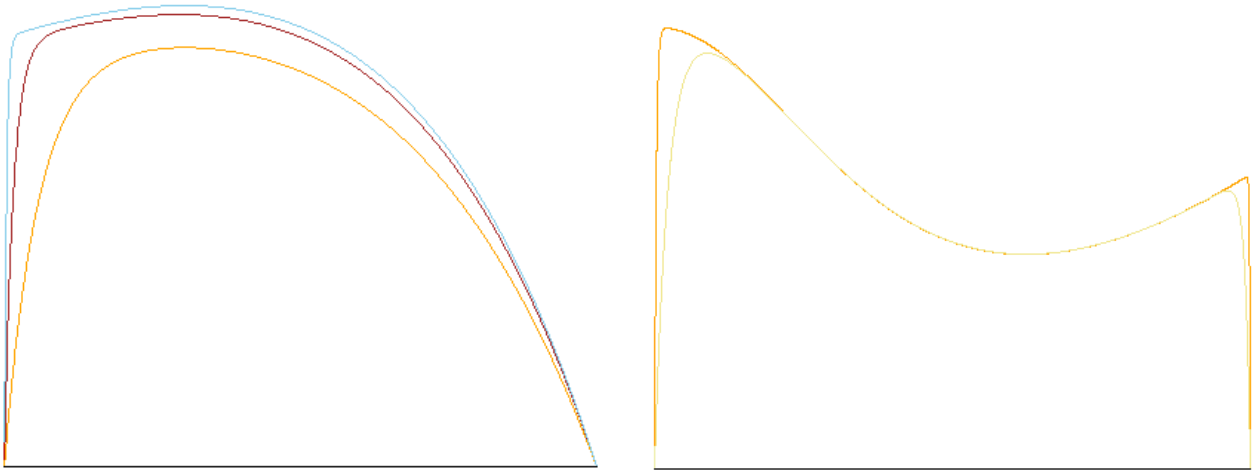

Abbildung 1: Schematische Darstellung typischen Grenzschichtverhaltens bei Reaktionsdiffusionsproblemen (rechts) und Konvektionsdiffusionsproblemen (links).

2. Reaktionsdiffusionsprobleme:

$$-\varepsilon y'' + qy = f, \quad q(x) \geq \gamma^2, \quad x \in [s,t],$$

für ein $\gamma > 0$. Bei solchen Gleichungen tritt typischerweise je eine Grenzschicht an beiden Enden des betrachteten Intervalls auf, deren Verhalten durch die Grenzschichtfunktionen $e^{\frac{-\gamma x}{\varepsilon}}$ und $e^{\frac{-\gamma(1-x)}{\varepsilon}}$ beschrieben werden kann. Eine detaillierte Analyse des allgemeinen Verhaltens von Problemen dieser Art findet sich in Abschnitt 3.3.1 von [11].

Das charakteristische Verhalten der Lösungen beider Problemklassen stellen wir in Abbildung 1 dar.

3.2 Beschreibung der verwendeten Methode

Bei der Konstruktion des zur Näherung verwendeten Verfahrens, folgen wir im Wesentlichen [5], wo allgemeinere, klassische Probleme untersucht werden. Wir werden einen C^1-Spline $g \in \Pi_{k+1,\xi,1}$ als Näherung für die Lösung finden. Dazu wählen wir für $l \in \mathbb{N}$ eine streng monoton wachsende Gitterknotenfolge $\xi := (\xi_i)_{i=0}^{l}, \xi \subset I$, mit $\xi_0 = s, \xi_l = t$. Anschließend wählen wir eine Menge von Kollokationspunkten

$$\Delta := \{\tau_1, \ldots, \tau_{kl}\} \subset (s,t), \quad \tau_1 < \ldots < \tau_{kl},$$

für ein $k \in \mathbb{N}$, von denen genau k in jeweils einem Gitterintervall $(\xi_i, \xi_{i+1}), i = 0, \ldots, l-1$, liegen und die in allen Gitterintervallen auf die gleiche Weise verteilt werden. Schließlich setzen wir dann basierend auf den Bernstein-Polynomen ${}^{\xi_{i+1}}_{\xi_i}B_0^{k+1}, \ldots, {}^{\xi_{i+1}}_{\xi_i}B_{k+1}^{k+1}, i = 0, \ldots, l-1$, auf jedem Gitterintervall eine Funktion $g_i \in \Pi_{k+1}$ an, die in den Gitterknoten C^1-stetig an die angrenzenden Funktionen anschließt. Die Koeffizienten ermitteln wir durch Lösen eines linearen Gleichungssystems, welches sich aus den zwei Rand-, den $l \cdot k$ Kollokations- und den $2(l-1)$ Stetigkeitsbedingungen zusammensetzt. Der Einfachheit halber sei in den nächsten Abschnitten die Gitterknotenfolge beliebig. In Abschnitt 3.4 gehen wir diesbezüglich ins Detail und wählen für alle auftretenden Fälle bestimmte Gitter aus.

3.2.1 Wahl der Kollokationspunkte

Zur Bestimmung der Kollokationspunkte wählen wir $-1 < \rho_1 < \ldots < \rho_k < 1$ und setzen dann

$$\tau_{ik+j} := \frac{\xi_i + \xi_{i+1} + \rho_j(\xi_{i+1} - \xi_i)}{2}, \quad i = 0, \ldots, l-1, \quad j = 1, \ldots, k.$$

So liegen in jedem Gitterintervall k Kollokationspunkte, und diese sind immer auf die gleiche Weise verteilt. Die erhaltenen Zerlegungen sind lediglich abhängig von den $\rho_j, j = 1, \ldots, k$, welche wir dann nach unseren Anforderungen und Erkenntnissen bestmöglich wählen können. In Abschnitt 4 werden wir die Festlegung auf Gauss-Legendre-Punkte ausführlich begründen.

3.2.2 Der Ansatz

Die gesuchte Näherung setzen wir nach den Definitionen 2.1.8 für Polynome in Bézier-Darstellung und 2.2.5 für Splines als durch die Funktionen

$$g_i(x) = \sum_{j=0}^{k+1} \xi_i^{\xi_{i+1}} B_j^{k+1}(x) \cdot b_{ij} \in \Pi_{k+1}, \quad x \in [\xi_i, \xi_{i+1}], \quad i = 0, \ldots, l-1,$$

definierten C^1-Bézier-Spline an.

3.3 Bestimmung des Gleichungssystems

Im Folgenden gilt, wenn nichts anderes angegeben ist, $i = 0, \ldots, l-1$, und $j = 1, \ldots, k$. Wir bezeichnen $[\xi_i, \xi_{i+1}]$ als das i-te Gitterintervall, beziehungsweise τ_{ik+j} als den j-ten Kollokationspunkt des i-ten Gitterintervalls. Mit dem Algorithmus 2.1.9 von de Casteljau lassen sich sowohl die Funktions- als auch nach Satz 2.1.10 mit Lemma 2.1.11 die Ableitungswerte von g rekursiv in Abhängigkeit von den Bézier-Punkten bestimmen und so zusammen mit $p, q, f \in C(I)$ die $l \cdot k$ Kollokationsbedingungen (K_{ij}) formulieren. Die $2(l-1)$ Stetigkeitsbedingungen $(S_{10}, S_{11}, S_{20}, \ldots, S_{l-2,1}, S_{l-1,0}, S_{l-1,1})$ in den Gitterknoten $\xi_1, \ldots, \xi_{l-1}$ für den C^0- und den C^1-stetigen Übergang angrenzender Gitterintervalle, sowie die zwei Randbedingungen (R_s und R_t), ergeben sich aus Korollar 2.1.12. Bringen wir alle Gleichungen in die Reihenfolge, in der sie mit den Bézier-Punkten der einzelnen Gitterintervalle in Berührung kommen, ergibt sich ein fast blockdiagonales, lineares Gleichungssystem, das schematisch mit den soeben genannten

Bezeichnungen folgende Zeilen enthält

$$\begin{pmatrix} R_s & & & & & & & & \\ K_{01} & & & & & & & & \\ \vdots & & & & & & & & \\ K_{0k} & & & & & & & & \\ & S_{11} & & & & & & & \\ & S_{10} & & & & & & & \\ & & K_{11} & & & & & & \\ & & \vdots & & & & & & \\ & & K_{1k} & & & & & & \\ & & & S_{21} & & & & & \\ & & & & \ddots & & & & \\ & & & & & S_{l-1,0} & & & \\ & & & & & & K_{l-1,1} & & \\ & & & & & & \vdots & & \\ & & & & & & K_{l-1,k} & & \\ & & & & & & R_t & & \end{pmatrix} \begin{pmatrix} b_{00} \\ b_{01} \\ \vdots \\ b_{0k} \\ b_{0,k+1} \\ b_{10} \\ b_{11} \\ \vdots \\ b_{l-2,k} \\ b_{l-2,k+1} \\ b_{l-1,0} \\ b_{l-1,1} \\ \vdots \\ b_{l-1,k} \\ b_{l-1,k+1} \end{pmatrix} = \begin{pmatrix} R_s \\ K_{01} \\ \vdots \\ K_{0k} \\ S_{11} \\ S_{10} \\ K_{11} \\ \vdots \\ K_{1k} \\ S_{21} \\ \ddots \\ S_{l-1,0} \\ K_{l-1,1} \\ \vdots \\ K_{l-1,k} \\ R_t \end{pmatrix},$$

mit der $l(k+2) \times l(k+2)$ Koeffizientenmatrix $A = (\alpha_{mn})_{m,n=0}^{l(k+2)-1}$ für den Vektor der BÉZIER-Punkte $b_{00}, \ldots, b_{l-1,k+1}$. Es ergibt sich also jeweils für die BÉZIER-Punkte des 0-ten und $(l-1)$-ten Gitterintervalls ein Block mit $k+3$ Zeilen und für alle anderen Gitterintervalle Blöcke mit je $k+4$ Zeilen, wobei die jeweils ersten und letzten beiden Zeilen zweier aneinandergrenzender Blöcke die Stetigkeitsbedingungen enthalten und sich die Blöcke insofern dort überschneiden.

3.3.1 Die Randbedingungen

Die Randbedingungen und damit die erste und letzte Gleichung unseres linearen Gleichungssystems ergeben sich aus Korollar 2.1.12, so dass gilt

$$y_s = y(s) = g(s) = b_{00},$$

und

$$y_t = y(t) = g(t) = b_{l-1,k+1}.$$

3.3.2 Die Kollokationsbedingungen

Als Kollokationsbedingungen setzen wir an

$$-\varepsilon g_i''(\tau_{ik+j}) - p_{ik+j}\, g_i'(\tau_{ik+j}) + q_{ik+j}\, g_i(\tau_{ik+j}) = f_{ik+j},$$

bestimmen also g, so dass die Differentialgleichung in den Kollokationspunkten $\tau_1, \ldots, \tau_{lk}$ erfüllt ist. Dazu bestimmen wir zunächst die Funktions- und mit dem Algorithmus 2.1.9 von DE CASTELJAU die Ableitungswerte unserer Näherungslösung und anschließend daraus die Einträge unserer Koeffizientenmatrix. Aus den Definitionen 2.1.3 der BERNSTEIN-Polynome und 2.1.8 von Funktionen in BÉZIER-Darstellung folgt für die Funktionswerte unmittelbar

$$g_i(x) = \sum_{j=0}^{k+1} \binom{k+1}{j} (1-\mu)^{k+1-j}\, \mu^j\, b_{ij}.$$

Für die ersten Ableitungen gilt wegen Satz 2.1.10

$$
\begin{aligned}
g_i'(x) &= \frac{k+1}{\xi_{i+1}-\xi_i}\left(b_{i1}^k(x) - b_{i0}^k(x)\right) \\
&= \frac{k+1}{\xi_{i+1}-\xi_i}\left(\sum_{j=1}^{k+1}\binom{k}{j-1}(1-\mu)^{k-j+1}\mu^{j-1}b_{ij} - \sum_{j=0}^{k}\binom{k}{j}(1-\mu)^{k-j}\mu^j b_{ij}\right), \text{ Lemma 2.1.11,} \\
&= \frac{k+1}{\xi_{i+1}-\xi_i}\left(\mu^k b_{i,k+1} + \sum_{j=1}^{k}\left(\binom{k}{j-1}(1-\mu)^{k-j+1}\mu^{j-1} - \binom{k}{j}(1-\mu)^{k-j}\mu^j\right)b_{ij}\right. \\
&\quad \left. - (1-\mu)^k b_{i0}\right) \\
&= \frac{k+1}{\xi_{i+1}-\xi_i}\left(\mu^k b_{i,k+1} + \sum_{j=1}^{k}\left(\binom{k}{j-1}(1-\mu) - \binom{k}{j}\mu\right)(1-\mu)^{k-j}\,\mu^{j-1}\,b_{ij}\right. \\
&\quad \left. - (1-\mu)^k b_{i0}\right) \\
&= \frac{k+1}{\xi_{i+1}-\xi_i}\left(\mu^k b_{i,k+1} + \sum_{j=1}^{k}\left(\binom{k}{j-1} - \binom{k}{j-1}\mu - \binom{k}{j}\mu\right)(1-\mu)^{k-j}\,\mu^{j-1}\,b_{ij}\right. \\
&\quad \left. - (1-\mu)^k b_{i0}\right) \\
&= \frac{k+1}{\xi_{i+1}-\xi_i}\left(\mu^k b_{i,k+1} + \sum_{j=1}^{k}\left(\binom{k}{j-1} - \binom{k+1}{j}\mu\right)(1-\mu)^{k-j}\,\mu^{j-1}\,b_{ij}\right. \\
&\quad \left. - (1-\mu)^k b_{i0}\right) \text{ mit (1),}
\end{aligned}
$$

und für die zweiten Ableitungen wiederum wegen Satz 2.1.10

$$
\begin{aligned}
g_i''(x) &= \frac{(k+1)k}{(\xi_{i+1}-\xi_i)^2}\sum_{j=0}^{2}\binom{2}{j}(-1)^{2-j}b_{ij}^{k-1}(x) = \frac{(k+1)k}{(t-s)^2}\left(b_{i2}^{k-1} - 2b_{i1}^{k-1} + b_{i0}^{k-1}\right) \\
&= \frac{(k+1)k}{(\xi_{i+1}-\xi_i)^2}\left(\sum_{j=2}^{k+1}\binom{k-1}{j-2}(1-\mu)^{k+1-j}\,\mu^{j-2}\,b_{ij} - 2\sum_{j=1}^{k}\binom{k-1}{j-1}(1-\mu)^{k-j}\,\mu^{j-1}\,b_{ij}\right. \\
&\quad \left. + \sum_{j=0}^{k-1}\binom{k-1}{j}(1-\mu)^{k-1-j}\,\mu^j\,b_{ij}\right), \text{ mit Lemma 2.1.11,} \\
&= \frac{(k+1)k}{(\xi_{i+1}-\xi_i)^2}\left(\mu^{k-1}\,b_{i,k+1} + \Big((k-1)(1-\mu) - 2\mu\Big)\,\mu^{k-2}\,b_{ik}\right. \\
&\quad + \sum_{j=2}^{k-1}\left(\binom{k-1}{j-2}(1-\mu)^2 - 2\binom{k-1}{j-1}(1-\mu)\,\mu + \binom{k-1}{j}\mu^2\right)(1-\mu)^{k-1-j}\,\mu^{j-2}\,b_{ij} \\
&\quad \left. + \Big((k-1)\mu - 2(1-\mu)\Big)(1-\mu)^{k-2}\,b_{i1} + (1-\mu)^{k-1}\,b_{i0}\right).
\end{aligned}
$$

Damit folgt für die Einträge bezüglich b_{i0} der Koeffizientenmatrix, welche aus der j-ten Kollokationsbedingung des i-ten Blocks hervorgehen

$$\begin{aligned}
\alpha_{i(k+2)+j,i(k+2)} &= (-\varepsilon)\frac{(k+1)k}{(\xi_{i+1}-\xi_i)^2}(1-\mu_{ik+j})^{k-1} + p_{ik+j}\,\frac{k+1}{\xi_{i+1}-\xi_i}(1-\mu_{ik+j})^k \\
&\quad + q_{ik+j}\,(1-\mu_{ik+j})^{k+1} \\
&= (1-\mu_{ik+j})^{k-1}\left(p_{ik+j}\,\frac{k+1}{\xi_{i+1}-\xi_i}(1-\mu_{ik+j})\right. \\
&\quad \left. + q_{ik+j}\,(1-\mu_{ik+j})^2 - \frac{\varepsilon(k+1)k}{(\xi_{i+1}-\xi_i)^2}\right),
\end{aligned}$$

und analog aus den Koeffizienten von $b_{i,k+1}$

$$\begin{aligned}
\alpha_{ik+j,(i+1)(k+2)-1} &= (-\varepsilon)\frac{(k+1)k}{(\xi_{i+1}-\xi_i)^2}\mu_{ik+j}^{k-1} - p_{ik+j}\,\frac{k+1}{\xi_{i+1}-\xi_i}\,\mu_{ik+j}^{k} + q_{ik+j}\,\mu_{ik+j}^{k+1} \\
&= \mu_{ik+j}^{k-1}\left(q_{ik+j}\,\mu_{ik+j}^2 - p_{ik+j}\,\frac{k+1}{\xi_{i+1}-\xi_i}\,\mu_{ik+j} - \frac{\varepsilon(k+1)k}{(\xi_{i+1}-\xi_i)^2}\right)
\end{aligned}$$

sowie aus den Summanden, die b_{i1} und b_{ik} enthalten

$$\begin{aligned}
\alpha_{i(k+2)+j,i(k+2)+1} &= (-\varepsilon)\frac{(k+1)k}{(\xi_{i+1}-\xi_i)^2}\Big((k-1)\mu_{ik+j} - 2(1-\mu_{ik+j})\Big)(1-\mu_{ik+j})^{k-2} - p_{ik+j} \\
&\quad \frac{k+1}{\xi_{i+1}-\xi_i}(1-(k+1)\mu_{ik+j})(1-\mu_{ik+j})^{k-1} + q_{ik+j}\,(k+1)(1-\mu_{ik+j})^k\,\mu_{ik+j} \\
&= (k+1)(1-\mu_{ik+j})^{k-2}\left(\frac{\varepsilon k}{(\xi_{i+1}-\xi_i)^2}\Big(2(1-\mu_{ik+j}) - (k-1)\mu_{ik+j}\Big)\right. \\
&\quad \left. - p_{ik+j}\,\frac{1}{\xi_{i+1}-\xi_i}(1-(k+1)\mu_{ik+j})(1-\mu_{ik+j}) + q_{ik+j}\,(1-\mu_{ik+j})^2\,\mu_{ik+j}\right), \\
&= (k+1)(1-\mu_{ik+j})^{k-2}\left(\frac{\varepsilon k}{(\xi_{i+1}-\xi_i)^2}\Big(2-(k+1)\mu_{ik+j}\Big)\right. \\
&\quad \left. - p_{ik+j}\,\frac{1}{\xi_{i+1}-\xi_i}(1-(k+1)\mu_{ik+j})(1-\mu_{ik+j}) + q_{ik+j}\,(1-\mu_{ik+j})^2\,\mu_{ik+j}\right), \\
\alpha_{i(k+2)+j,i(k+2)+k} &= (-\varepsilon)\frac{(k+1)k}{(\xi_{i+1}-\xi_i)^2}\Big((k-1)(1-\mu_{ik+j}) - 2\mu_{ik+j}\Big)\mu_{ik+j}^{k-2} \\
&\quad - p_{ik+j}\,\frac{k+1}{\xi_{i+1}-\xi_i}(k-(k+1)\mu_{ik+j})\mu_{ik+j}^{k-1} + q_{ik+j}\,(k+1)\mu_{ik+j}^{k}\,(1-\mu_{ik+j}) \\
&= (k+1)\mu_{ik+j}^{k-2}\left(\frac{\varepsilon k}{(\xi_{i+1}-\xi_i)^2}\Big(2\mu_{ik+j} - (k-1)(1-\mu_{ik+j})\Big)\right. \\
&\quad \left. - p_{ik+j}\,\frac{1}{\xi_{i+1}-\xi_i}(k-(k+1)\mu_{ik+j})\,\mu_{ik+j} + q_{ik+j}\,\mu_{ik+j}^2\,(1-\mu_{ik+j})\right) \\
&= (k+1)\mu_{ik+j}^{k-2}\left(\frac{\varepsilon k}{(\xi_{i+1}-\xi_i)^2}\Big((k+1)\mu_{ik+j} - k + 1\Big)\right. \\
&\quad \left. - p_{ik+j}\,\frac{1}{\xi_{i+1}-\xi_i}(k-(k+1)\mu_{ik+j})\,\mu_{ik+j} + q_{ik+j}\,\mu_{ik+j}^2\,(1-\mu_{ik+j})\right).
\end{aligned}$$

Für die Koeffizienten aller anderen BÉZIER-Punkte folgt

$$\begin{aligned}\alpha_{i(k+2)+j,i(k+2)+\kappa} = \frac{\varepsilon(k+1)k}{(\xi_{i+1}-\xi_i)^2}\Bigg(&2\binom{k-1}{\kappa-1}(1-\mu_{ik+j})\,\mu_{ik+j} - \binom{k-1}{\kappa-2}(1-\mu_{ik+j})^2\\ &-\binom{k-1}{\kappa}\mu_{ik+j}^2\Bigg)(1-\mu_{ik+j})^{k-1-\kappa}\,\mu_{ik+j}^{\kappa-2} - p_{ik+j}\frac{k+1}{\xi_{i+1}-\xi_i}\\ &\left(\binom{k}{\kappa-1} - \binom{k+1}{\kappa}\mu_{ik+j}\right)(1-\mu_{ik+j})^{k-\kappa}\,\mu_{ik+j}^{\kappa-1}\\ &+ q_{ik+j}\binom{k+1}{\kappa}(1-\mu_{ik+j})^{k+1-\kappa}\,\mu_{ik+j}^{\kappa}, \quad \kappa = 2,\ldots,k-1.\end{aligned}$$

3.3.3 Die Stetigkeitsbedingungen

Die Stetigkeitsbedingungen in den Gitterknoten $\xi_i, i = 1,\ldots,l-1$, ergeben sich unmittelbar aus Korollar 2.1.12. Für einen C^0-stetigen Übergang gilt dort

$$b_{i-1,k+1} = g_{i-1}(\xi_i) = g_i(\xi_i) = b_{i0}, \tag{5}$$

und aus der geforderten C^1-Stetigkeit folgt

$$\begin{aligned}\frac{k+1}{\xi_i-\xi_{i-1}}\left(b_{i-1,k+1}-b_{i-1,k}\right) = \qquad & g'_{i-1}(\xi_i) = g'_i(\xi_i) \qquad = \frac{k+1}{\xi_{i+1}-\xi_i}\left(b_{i1}-b_{i0}\right),\\ \frac{1}{\xi_i-\xi_{i-1}}\left(b_{i-1,k+1}-b_{i-1,k}\right) & \qquad = \frac{1}{\xi_{i+1}-\xi_i}\left(b_{i1}-b_{i0}\right),\end{aligned}$$

also

$$\begin{aligned}(b_{i-1,k+1}-b_{i-1,k})(\xi_{i+1}-\xi_i) &= (b_{i1}-b_{i0})(\xi_i-\xi_{i-1})\\ b_{i-1,k+1}(\xi_{i+1}-\xi_i) &= (b_{i1}-b_{i-1,k+1})(\xi_i-\xi_{i-1}) + b_{i-1,k}(\xi_{i+1}-\xi_i), \text{ mit (5)}\\ b_{i-1,k+1}(\xi_{i+1}-\xi_{i-1}) &= b_{i1}(\xi_i-\xi_{i-1}) + b_{i-1,k}(\xi_{i+1}-\xi_i), \text{ für } i = 1,\ldots,l-1.\end{aligned}$$

Bei äquidistanten Gitterknoten ergibt sich demnach

$$2\cdot b_{i-1,k+1} = b_{i1} + b_{i-1,k}, \quad i = 1,\ldots,l-1.$$

3.3.4 Lösen des Gleichungssystems

Mit einem geeigneten Verfahren lässt sich dann die eindeutige Näherungslösung für das Randwertproblem ermitteln. Wir verwenden hierzu den GAUSS-Algorithmus basierend auf einer besonders speicherarmen Version des Algorithmus `bandet1` aus [13] zur Lösung linearer Gleichungssysteme mit symmetrischen und asymmetrischen Bandmatrizen. Näheres zur Implementierung findet sich im Abschnitt 6.

3.4 Wahl der Gitterknoten

Bei der Wahl der Gitterknoten definieren wir zunächst für den klassischen Fall $\bar{\xi} := (\bar{\xi}_i)_{i=0}^{l}$ mit

$$\bar{\xi}_i = s + i\frac{t-s}{l}, i = 0,\ldots,l,$$

also ein über dem gesamten Intervall $[s,t]$ uniformes Gitter. Bei der Untersuchung des singulär gestörten Falls, stellen sich äquidistante Gitterknoten als unzureichend heraus, um im Allgemeinen nützliche Aussagen über die Güte der Näherung zu treffen. Deshalb definieren wir zusätzlich zwei sogenannte a priori grenzschichtangepasste Gitter gemäß Kapitel 2 in [11] und Abschnitt 2.2 in [9]. A priori bedeutet, dass die Gitter vor der Durchführung des Verfahrens festgelegt und im Verlauf nicht mehr geändert werden. Beide Definitionen teilen das untersuchte Intervall in feine Gitter innerhalb und grobe Gitter außerhalb der Grenzschicht(en). Das SHISHKIN-Gitter erzeugt eine stückweise äquidistante Folge von Gitterknoten und das BAKHVALOV-Gitter die Projektion eines äquidistanten Gitters. Die Bestimmung der Gitterknoten erfolgt schließlich mittels einer gittererzeugenden Funktion

Definition 3.4.1. *Eien streng monotone Funktion $\varphi\colon [0,1] \to [0,1]$, welche das uniforme Gitter $r_i = \frac{i}{l}, i = 0,\dots,l$, auf ein grenzschichtangepasstes Gitter $\xi_i := \varphi(t_i), i = 0,\dots,l$, abbildet, heißt gittererzeugende Funktion,*

oder bei uniformen Gittern und analog uniformen Teilgittern auch durch

$$\xi_i = \xi_{i-1} + h, \quad i = 1,\dots,l.$$

Seien im Folgenden $q, q_0, q_1 \in (0,1), q_0 + q_1 \leq 1$, und $\sigma, \sigma_0, \sigma_1 > 0$ frei wählbare Gitterparameter. Wir betrachten zunächst die Gitterkonstruktion bei Auftreten einer Grenzschicht, also Konvektionsdiffusionsprobleme, und übetragen die Betrachtungen anschließend auf Reaktionsdiffusionsprobleme.

Definition 3.4.2 (SHISHKIN-Gitter bei Konvektionsdiffusionsproblemen)**.** *Wir definieren einen Gitterübergangspunkt τ durch*

$$\tau := \min\left\{q, \frac{\sigma\varepsilon}{\beta}\ln l\right\}.$$

Anschließend teilen wir $[s,t]$ in $[s, s+\tau(t-s)], [s+\tau(t-s), t]$. Diese Teilgitter werden anschließend in $\lfloor ql \rfloor$ und $l - \lfloor ql \rfloor$ äquidistante Gitterintervalle zerlegt. Wir erhalten $\xi^S := (\xi_i^S)_{i=0}^l$, mit

$$\xi_i^S - \xi_{i-1}^S = \begin{cases} \frac{\tau(t-s)}{\lfloor ql \rfloor} & \text{, falls } 0 < i \leq ql, \\ \frac{(1-\tau)(t-s)}{l - \lfloor ql \rfloor} & \text{, falls } ql < i \leq l. \end{cases}$$

Die gittererzeugende Funktion hat die Gestalt

$$\varphi(r) = \begin{cases} \frac{\sigma\varepsilon}{\beta}\frac{r}{q}\ln l & \text{, falls } r \in [0,q], \\ 1 - \left(1 - \frac{\sigma\varepsilon}{\beta}\ln l\right)\frac{1-r}{1-q} & \text{, falls } r \in [q,1]. \end{cases}$$

Definition 3.4.3 (BAKHVALOV-Gitter bei Konvektionsdiffusionsproblemen)**.** *Wir definieren die Gitterpunkte $\xi_i^B := s + z_i(t-s), 0 \leq i \leq l$, in der Grenzschicht durch*

$$\begin{aligned} q\left(1 - e^{\frac{-\beta z_i}{\sigma\varepsilon}}\right) = r_i = \frac{i}{l} \quad & \text{für } i = 0,1,\dots, \\ \Rightarrow \qquad z_i = -\frac{\sigma\varepsilon}{\beta}\ln\frac{q - r_i}{q}, & \end{aligned}$$

wie üblich zunächst auf dem Intervall $[0,1]$ und transformieren anschließend auf unser Intervall $[s,t]$. Außerhalb der Grenzschicht definieren wir die ξ_i^B als äquidistant und erhalten die gittererzeugende Funktion

$$\varphi(r) = \begin{cases} \chi(r) := \frac{-\sigma\varepsilon}{\beta}\ln\frac{q-r}{q} & \text{, falls } r \in [0,\tau], \\ \pi(r) := \chi(\tau) + \chi'(\tau)(r-\tau) & \text{, falls } r \in [\tau,1]. \end{cases}$$

Für $\tau < q$ ist φ im Intervall $[0,1]$ wegen

$$\pi'(\tau) = \chi'(\tau) = \frac{\sigma\varepsilon}{\beta(q-\tau)}, \tag{6}$$

sogar C^1-stetig. Dabei wählen wir τ so, dass die Tangente am Graphen von φ in τ durch den Punkt $(1,1)$ verläuft, also

$$\chi'(\tau) = \frac{1-\chi(\tau)}{1-\tau}. \tag{7}$$

Dadurch wird sichergestellt, dass sich ein das ganze Teilintervall $[\tau,1]$ überdeckendes, uniformes Teilgitter ergibt. Schließlich ist

$$\xi_i^B := s + \varphi(r_i)(t-s), \quad i = 0,\dots,l.$$

Da die Lösung der nichtlinearen Gleichung (7) nicht explizit bestimmt werden kann, verwenden wir die in Abschnitt 2.1.1. aus [11] vorgeschlagene Iteration

$$\tau_0 = 0, \quad \chi'(\tau_{i+1}) = \frac{1-\chi(\tau_i)}{1-\tau_i}, \quad i = 0,1,\dots, \tag{8}$$

zur Bestimmung von τ. Mit (6) erhalten wir

$$\begin{aligned}
\tau_{i+1} &= \frac{\beta q - \frac{\sigma\varepsilon}{\chi'(\tau_{i+1})}}{\beta} \\
&= q - \frac{\sigma\varepsilon}{\beta\chi'(\tau_{i+1})} \\
&= q - \frac{\sigma\varepsilon(1-\tau_i)}{\beta(1-\chi(\tau_i)}, \quad \text{mit (8)}, \quad i = 0,1,\dots.
\end{aligned}$$

Die Gitter im Falle des Auftretens zweier Grenzschichten haben folgende Gestalt.

Definition 3.4.4 (Shishkin-Gitter bei Reaktionsdiffusionsproblemen)**.** *Wir definieren zwei Gitterübergangspunkte*

$$\tau_0 := \min\left\{q_0, \frac{\sigma_0\varepsilon}{\gamma}\ln l\right\}, \tau_1 := \min\left\{q_1, \frac{\sigma_1\varepsilon}{\gamma}\ln l\right\}.$$

Anschließend teilen wir $[s,t]$ in drei Teilintervalle $[s, s+\tau_0(t-s)], [s+\tau_0(t-s), t-\tau_1(t-s)]$ und $[t-\tau_1(t-s), t]$. Diese Teilgitter werden dann in $\lfloor q_0 l\rfloor, l-(\lfloor q_0 l\rfloor + \lfloor q_1 l\rfloor)$ und $\lfloor q_1 l\rfloor$ äquidistante Gitterintervalle zerlegt. Wir erhalten diesmal $\xi^S := (\xi_i^S)_{i=0}^l$, mit

$$\xi_i^S - \xi_{i-1}^S = \begin{cases} \frac{\tau_0(t-s)}{\lfloor q_0 l\rfloor} & \text{, falls } 0 < i \le q_0 l, \\ \frac{(1-(\tau_0+\tau_1))(t-s)}{l-(\lfloor q_0 l\rfloor + \lfloor q_1 l\rfloor)} & \text{, falls } q_0 l < i \le l - \lfloor q_1 l\rfloor, \\ \frac{\tau_1(t-s)}{\lfloor q_1 l\rfloor} & \text{, falls } l - \lfloor q_1 l\rfloor < i \le l. \end{cases}$$

Definition 3.4.5 (Bakhvalov-Gitter bei Reaktionsdiffusionsproblemen)**.** *Wir definieren analog zu Definition 3.4.3 die Gitterpunkte $\xi_{0,i}^B := s + z_{0,i}(t-s), 0 \le i \le l$, in der Grenzschicht bei $x = s$ durch*

$$\begin{aligned}
q_0\left(1 - e^{\frac{-\gamma z_{0,i}}{\sigma_0\varepsilon}}\right) = r_i = \frac{i}{l} \quad &\text{für } i = 0,1,\dots, \\
\Rightarrow \qquad z_{0,i} &= -\frac{\sigma_0\varepsilon}{\gamma}\ln\frac{q_0 - r_i}{q_0},
\end{aligned}$$

und die Gitterpunkte $\xi_{1,i}^B := t - z_{1,i}(t-s), 0 \le i \le l$, *in der Grenzschicht bei* $x = t$ *durch*

$$1 - q_1\left(1 - e^{\frac{-\gamma(1-z_{1,i})}{\sigma_1 \varepsilon}}\right) = r_i = \frac{i}{l} \quad \text{für } i = l, l-1, \ldots,$$

$$\Rightarrow \qquad z_{1,i} = 1 + \frac{\sigma_1 \varepsilon}{\gamma} \ln \frac{q_1 + r_i - 1}{q_1}.$$

Außerhalb der Grenzschichten definieren wir die ξ_i^B *als äquidistant und erhalten die gittererzeugende Funktion*

$$\varphi(r) = \begin{cases} \chi_0(r) := \frac{\sigma_0 \varepsilon}{\gamma}\left(-\ln \frac{q_0 - r}{q_0}\right) & \text{, falls } r \in [0, \tau_0], \\ \pi(r) := \chi_0(\tau_0) + \chi_0'(\tau_0)(r - \tau_0) & \text{, falls } r \in [\tau_0, 1-\tau_1], \\ \chi_1(r) := 1 + \frac{\sigma_1 \varepsilon}{\gamma} \ln \frac{q_1 + r - 1}{q_1} & \text{, falls } r \in [1-\tau_1, 1]. \end{cases}$$

Dabei wählen wir τ_0 *und* τ_1 *so, dass die Tangente am Graphen von* χ_0 *in* τ_0 *mit der Tangente am Graphen von* χ_1 *in* $1 - \tau_1$ *übereinstimmt und somit das mittlere Teilgitter wiederum das ganze Intervall* $[\tau_0, 1 - \tau_1]$ *uniform überdeckt. Schließlich ist*

$$\xi_i^B := s + \varphi(r_i)(t - s), \quad i = 0, \ldots, l.$$

4 Konvergenzeigenschaften

Die Konvergenzeigenschaften des Verfahrens im klassischen und im singulär gestörten Fall betrachten wir getrennt, da sich die Eigenschaften des klassischen Falls im Allgemeinen nicht auf den singulär gestörten Fall übertragen lassen.

4.1 Der klassische Fall

Zunächst geben wir das zentrale Ergebnis aus [5] wieder, in dem der klassische Fall wesentlich allgemeiner untersucht wird, als unsere Arbeit das erfordert. Das dort untersuchte Problem ist

Aufgabe 4.1.1. *Finde* $x \in C^{(m)}[a,b], m > 0$, *so dass*

$$D^m x = F(t, x(t), \ldots, (D^{m-1}x)(t)), \qquad \beta_i x = c_i, \quad i = 1, \ldots, m.$$

Dort wird also für ein $m \in \mathbb{N}$ die in einem Intervall $[a,b]$ (mindestens) C^m-stetige Lösung einer nichtlinearen Differentialgleichung m-ter Ordnung gesucht, welche $m-1$ lineare Zusatzbedingungen erfüllt. Dementsprechend gelten für die dort als Satz 4.1 formulierte Aussage allgemeinere Voraussetzungen, die teils für unseren betrachteten Fall trivialerweise erfüllt sind. Es ergibt sich somit folgender Satz.

Satz 4.1.2. *Seien* $r \in \mathbb{N}, r \le k$, $f, q, p \in C^{\max(2,k+r)}[s,t]$ *und liege die Lösung unseres Randwertproblems in* $C^{(k+2+r)}[s,t]$. *Wenn wir die den Kollokationspunkten zugrundeliegenden* $(\rho_j)_{j=1}^k \in (-1,1)$ *so wählen, dass*

$$\int_{-1}^{1} q(\lambda) \prod_{j=1}^{k} (\lambda - \rho_j) \mathrm{d}\lambda = 0,$$

für alle $q \in \Pi_{r-1}$, *dann existiert eine vom Gitter* ξ *unabhängige Konstante* C, *so dass für die beste Näherung der* ν*-ten Ableitung in den Gitterknoten* $\xi_i, i = 0, \ldots, l$, *gilt*

$$|D^\nu (g - u)(x)| \le C h^{k+r}, \quad x \in \xi, \quad \nu = 0, \ldots, m-1.$$

Für die global beste Näherung gilt

$$\|D^\nu(g-u)\|_\infty \leq Ch^{k+\min(r,2-\nu)}, \quad \nu = 0,\ldots,m.$$

Wie in Abschnitt 3.2.1 angekündigt, wollen wir als die den Kollokationspunkten zugrundeliegenden $(\rho_j)_{j=1}^k$ die Nullstellen des k-ten LEGENDRE-Polynoms verwenden, welche auch GAUSS-LEGENDRE-Punkte genannt werden (siehe Kapitel 7 in [12]). Näheres zur konkreten Bestimmung findet sich in Abschnitt 6.2. Unter Berücksichtigung von Korollar 2.3.6 erreichen wir so jeweils die bestmögliche Konvergenzordnung mit $r = k$. Für das Verfahren auf unser Problem angewendet ergibt sich damit für die beste Näherung in den Gitterknoten

$$|D^\nu(g-u)(x)| \leq Ch^{2k}, \quad x \in \xi, \quad \nu = 0,1, \tag{9}$$

und für die global beste Näherung

$$\|D^\nu(g-u)\|_\infty \leq Ch^{k+\min(k,2-\nu)}, \quad \nu = 0,1,2. \tag{10}$$

4.2 Der singulär gestörte Fall

Wie eingangs erwähnt, lassen sich die Eigenschaften des klassischen Falls nicht ohne Weiteres auf den singulär gestörten Fall übertragen, da im Allgemeinen für $\varepsilon \to 0$ die Konstante C über alle Maße wächst. Wir geben deshalb bezüglich der Konvergenzeigenschaften im singulär gestörten Fall die zu erwartenden Konvergenzordnungen für die zwei erprobten a priori grenzschichtangepassten Gitter gemäß der Definitionen in Abschnitt 3.4 an. Dabei stützen wir uns auf die grundlegenden Ausführungen aus Kapitel 2 in [11] und geben die Ergebnisse in der dort dargestellten, einfachen Form wieder. Es bezeichne u die exakte Lösung des Problems und u^l die durch unser Verfahren mit l Gitterintervallen ermittelte Näherung.

Bemerkung 4.2.1. *Für die Anwendung unseres Verfahrens ist im singulär gestörten Fall mit folgenden Fehlerschranken zu rechnen.*

- *Bei Verwendung eines* BAKHVALOV-*Gitters*

$$\|u-u^l\|_\infty \leq K(l^{-1})^\alpha,$$

- *Bei Verwendung eines* SHISHKIN-*Gitters*

$$\|u-u^l\|_\infty \leq K(l^{-1}\ln l)^\alpha,$$

jeweils für ein $\alpha > 0$ und eine von ε unabhängige Konstante K.

5 Numerische Beispiele

Zur exprimentellen Bestätigung der theoretischen Ergebnisse behandeln wir ein häufig in der Literatur vorzufindendes Beispiel für den klassischen Fall und anschließend zwei singulär gestörte Probleme.

Tabelle 1: Zu erwartende Konvergenzordnungen von Beispiel 5.1.1 nach Satz 4.1.2.

	α_∞			$\alpha_{\bar\xi}$		
	k					
ν	1	2	4	1	2	4
0	2	4	6	2	4	8
1	2	3	5	2	4	8
2	1	2	4	2	4	8

5.1 Der klassische Fall

Zur Verifizierung der Güte unserer Näherungslösung im klassischen Fall betrachten wir ein Beispiel aus [5] und [1].

Beispiel 5.1.1.

$$y''(x) - 4y(x) = 4\cosh(1), \quad y(0) = y(1) = 0,$$

mit der exakten Lösung

$$u(x) = \cosh(2x - 1) - \cosh(1).$$

Die nach Satz 4.1.2 zu erwartenden Ordnungen für die globale Konvergenz der Näherung geben wir in Tabelle 1 wieder. Dargestellt werden die Werte

$$\alpha_\infty := k + \min(k, 2 - \nu), \qquad \alpha_{\bar\xi} := 2k.$$

Wir stellen in den Tabellen 2, 3 und 4 für verschiedene Werte von $k, l \in \mathbb{N}$ die Fehler

$$E^{(\nu)}_{\infty,l} := \|g^{(\nu)} - u^{(\nu)}\|_\infty,$$
$$E^{(\nu)}_{\bar\xi,l} := \|g^{(\nu)} - u^{(\nu)}\|_{\bar\xi} = \max_{i=0,\dots,l} |g^{(\nu)}(\bar\xi_i) - u^{(\nu)}(\bar\xi_i)|,$$

für $\nu = 0, 1, 2$, dar, wobei wir für $\nu = 2$ nach Satz 4.1.2 erwartungsgemäß $E^{(2)}_{\bar\xi,l} = E^{(2)}_{\infty,l}$ erhalten. Dabei ist $u\colon [s,t] \to \mathbb{R}$ die exakte Lösung der betreffenden Randwertaufgabe. Aufgrund der Einfachheit und ohne dadurch die Genauigkeit unserer dargestellten Ergebnisse zu verfälschen, bestimmen wir für den globalen Fehler die diskrete Maximumnorm für 13 in jedem der l Teilintervalle äquidistant verteilte Stellen. Zur Bestimmung der experimentellen Konvergenzordnung $\alpha_{\sigma,l}, \sigma \in \{\infty, \bar\xi\}$, stellen wir fest

$$E^{(\nu)}_{\sigma,l} \le Ch^{\alpha_\sigma} \qquad \text{, Gleichungen (9) und (10),}$$
$$E^{(\nu)}_{\sigma,l} \le \frac{C}{l^{\alpha_\sigma}} \qquad \text{, da wir ein uniformes Gitter verwenden,}$$
$$l^{\alpha_\sigma} E^{(\nu)}_{\sigma,l} \le C, \qquad l = 1, 2, \dots.$$

Deshalb setzen wir an

$$l^{\alpha_\sigma} E^{(\nu)}_{\sigma,l} \approx (l-1)^{\alpha_\sigma} E^{(\nu)}_{\sigma,l-1}$$
$$\frac{E^{(\nu)}_{\sigma,l}}{E^{(\nu)}_{\sigma,l-1}} \approx \frac{(l-1)^{\alpha_\sigma}}{l^{\alpha_\sigma}}$$
$$\ln \frac{E^{(\nu)}_{\sigma,l}}{E^{(\nu)}_{\sigma,l-1}} \approx \alpha_\sigma \ln \frac{l-1}{l}$$
$$\frac{\ln \frac{E^{(\nu)}_{\sigma,l}}{E^{(\nu)}_{\sigma,l-1}}}{\ln \frac{l-1}{l}} \approx \alpha_\sigma.$$

Tabelle 2: Fehler $E^{(0)}_{\bar{\xi},l}$ und $E^{(0)}_{\infty,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.1.1.

	k											
	1				2				4			
l	$E^{(0)}_{\infty,l}$	$\alpha_{\infty,l}$	$E^{(0)}_{\bar{\xi},l}$	$\alpha_{\bar{\xi},l}$	$E^{(0)}_{\infty,l}$	$\alpha_{\infty,l}$	$E^{(0)}_{\bar{\xi},l}$	$\alpha_{\bar{\xi},l}$	$E^{(0)}_{\infty,l}$	$\alpha_{\infty,l}$	$E^{(0)}_{\bar{\xi},l}$	$\alpha_{\bar{\xi},l}$
1	3,0e-2		0,0e-0		3,6e-2		0,0e-0		1,9e-04		0,0e-00	
2	1,8e-2	0,7	1,8e-2		2,7e-3	3,7	1,9e-4		3,5e-06	5,8	3,1e-09	
3	7,0e-3	2,3	6,8e-3	2,4	6,0e-4	3,7	3,3e-5	4,3	3,3e-07	5,8	1,1e-10	8,3
4	4,1e-3	1,9	4,1e-3	1,7	2,0e-4	3,8	1,1e-5	3,7	6,3e-08	5,8	1,2e-11	7,7
5	2,5e-3	2,1	2,5e-3	2,2	8,7e-5	3,8	4,4e-6	4,2	1,7e-08	5,8	1,9e-12	8,2
6	1,8e-3	1,9	1,8e-3	1,9	4,3e-5	3,9	2,2e-6	3,8	5,8e-09	5,9	4,6e-13	7,8
7	1,3e-3	2,1	1,3e-3	2,1	2,4e-5	3,9	1,2e-6	4,1	2,3e-09	5,9	1,3e-13	8,1
8	1,0e-3	1,9	1,0e-3	1,9	1,4e-5	3,9	6,9e-7	3,9	1,1e-09	5,9	4,6e-14	7,9
16	2,5e-4	2,0	2,5e-4	2,0	9,3e-7	3,9	4,3e-8	4,0	1,7e-11	5,9	1,8e-16	8,0
32	6,2e-5	2,0	6,2e-5	2,0	6,0e-8	4,0	2,7e-9	4,0	2,8e-13	6,0	7,0e-19	8,0

Tabelle 3: Fehler $E^{(1)}_{\bar{\xi},l}$ und $E^{(1)}_{\infty,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.1.1.

	k											
	1				2				4			
l	$E^{(1)}_{\infty,l}$	$\alpha_{\infty,l}$	$E^{(1)}_{\bar{\xi},l}$	$\alpha_{\bar{\xi},l}$	$E^{(1)}_{\infty,l}$	$\alpha_{\infty,l}$	$E^{(1)}_{\bar{\xi},l}$	$\alpha_{\bar{\xi},l}$	$E^{(1)}_{\infty,l}$	$\alpha_{\infty,l}$	$E^{(1)}_{\bar{\xi},l}$	$\alpha_{\bar{\xi},l}$
1	2,9e-1		2,9e-1		1,2e-1		3,6e-2		1,6e-3		1,5e-5	
2	1,1e-1	1,5	1,1e-1	1,5	1,8e-2	2,7	2,8e-3	3,7	5,6e-5	4,8	7,8e-8	7,6
3	5,0e-2	1,8	5,0e-2	1,8	5,8e-3	2,8	5,7e-4	3,9	8,1e-6	4,8	3,3e-9	7,8
4	3,0e-2	1,8	2,9e-2	1,9	2,6e-3	2,8	1,8e-4	3,9	2,0e-6	4,8	3,3e-10	7,9
5	2,1e-2	1,7	1,9e-2	1,9	1,4e-3	2,8	7,6e-5	4,0	6,8e-7	4,9	5,7e-11	7,9
6	1,5e-2	1,7	1,3e-2	2,0	8,1e-4	2,9	3,7e-5	4,0	2,8e-7	4,9	1,3e-11	8,0
7	1,2e-2	1,8	9,7e-3	2,0	5,2e-4	2,9	2,0e-5	4,0	1,3e-7	4,9	3,9e-12	8,0
8	9,0e-3	1,8	7,4e-3	2,0	3,5e-4	2,9	1,2e-5	4,0	6,8e-8	4,9	1,3e-12	8,0
16	2,5e-3	1,9	1,9e-3	2,0	4,6e-5	2,9	7,4e-7	4,0	2,2e-9	4,9	5,3e-15	8,0
32	6,5e-4	1,9	4,7e-4	2,0	5,9e-6	3,0	4,6e-8	4,0	7,0e-11	5,0	2,1e-17	8,0

In den Tabellen geben wir deshalb an

$$\alpha_{\sigma,l} := \ln \frac{E^{(\nu)}_{\sigma,l}}{E^{(\nu)}_{\sigma,l-1}} \Big/ \ln \frac{l-1}{l}. \tag{11}$$

Zur Veranschaulichung der Konvergenzeigenschaften haben wir die Graphen der Näherungslösungen $gkl, k, l = 1, 2$, zusammen mit dem Graphen der exakten Lösung in Abbildung 2 dargestellt.

5.2 Der singulär gestörte Fall

Die Konvergenzeigenschaften im singulär gestörten Fall, insbesondere die gleichmäßige Konvergenz bezüglich ε, beobachten wir jeweils ohne Kenntnis der exakten Lösungen mittels des folgenden Ansatzes aus [11]. Zu einer Näherungslösung u^l, die auf dem Gitter ξ berechnet wurde, wird eine Näherungslösung $\tilde{u}^l$ auf $\tilde{\xi}$, einem wesentlich feineren Gitter bestimmt. Dabei ergibt sich $\tilde{\xi}$, indem jedes Gitterintervall von ξ in sieben äquidistante Teilintervalle zerlegt wird. Für

Tabelle 4: Fehler $E^{(2)}_{\sigma,l}$, σ wie in (11), und experimentelle Konvergenzordnungen von Beispiel 5.1.1.

	k					
	1		2		4	
l	$E^{(2)}_{\sigma,l}$	$\alpha_{\sigma,l}$	$E^{(2)}_{\sigma,l}$	$\alpha_{\sigma,l}$	$E^{(2)}_{\sigma,l}$	$\alpha_{\sigma,l}$
1	2,1e-0		1,5e-0		4,1e-2	
2	1,7e-0	0,3	4,2e-1	1,9	2,9e-3	3,8
3	1,3e-0	0,7	2,0e-1	1,9	6,1e-4	3,8
4	1,0e-0	0,8	1,1e-1	1,9	2,0e-4	3,9
5	8,3e-1	0,9	7,5e-2	1,9	8,4e-5	3,9
6	7,0e-1	0,9	5,3e-2	1,9	4,1e-5	3,9
7	6,1e-1	0,9	3,9e-2	1,9	2,3e-5	3,9
8	5,4e-1	0,9	3,0e-2	1,9	1,3e-5	3,9
16	2,8e-1	0,9	7,8e-3	2,0	8,6e-7	4,0
32	1,4e-1	1,0	2,0e-3	2,0	5,5e-8	4,0

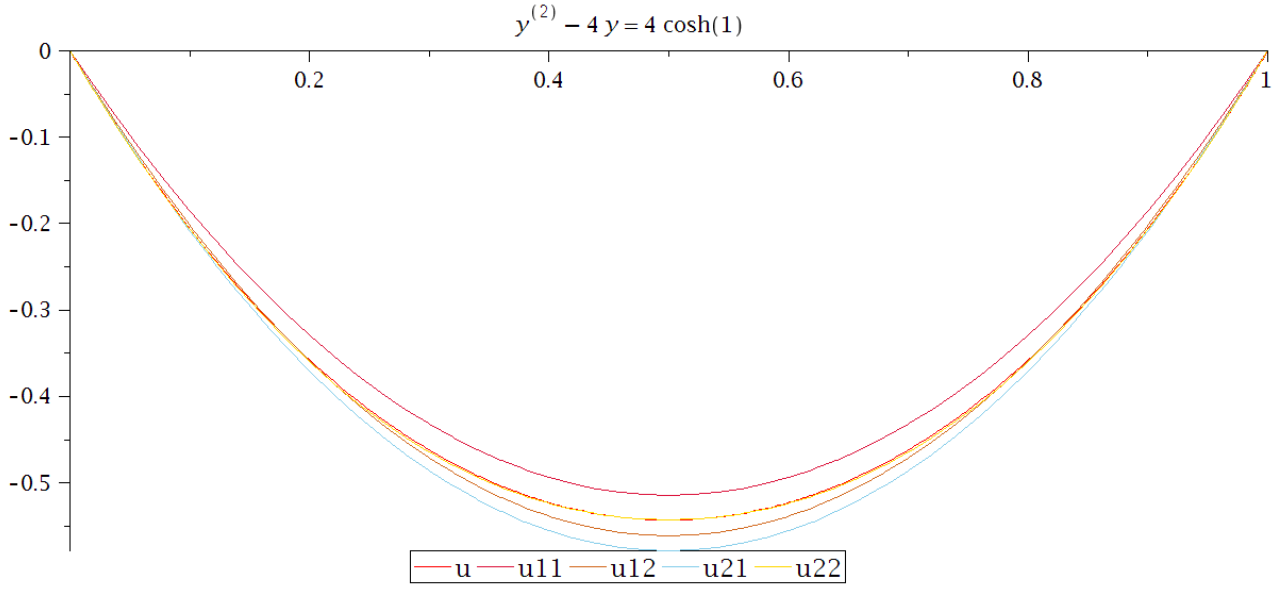

Abbildung 2: Die Graphen der exakten Lösung u und der Näherungslösungen von Beispiel 5.1.1 für $k, l = 1, 2$.

den Fehler E_ξ^l setzen wir dann an

$$\|u - u^l\|_\xi \approx E_{\xi,l} := \|\tilde{u}^l - u^l\|_\xi.$$

Diese Fehlernäherungen bestimmen wir für $l = 2^7, \ldots, 2^{12}$, so dass sich analog zu unseren Überlegungen für (11) unter Berücksichtigung von $K(l^{-1} \ln l)^\alpha$ als Fehlerschranke für SHISHKIN-Gitter ergibt

$$\begin{aligned}
\frac{E^S_{\xi,2l}}{((2l)^{-1}\ln 2l)^\alpha} &\approx \frac{E^S_{\xi,l}}{(l^{-1}\ln l)^\alpha} \\
\frac{E^S_{\xi,2l}}{E^S_{\xi,l}} &\approx \frac{((2l)^{-1}\ln 2l)^\alpha}{(l^{-1}\ln l)^\alpha} \\
\ln \frac{E^S_{\xi,2l}}{E^S_{\xi,l}} &\approx \alpha \ln \frac{\ln 2l}{2\ln l} \\
\frac{\ln \frac{E^S_{\xi,2l}}{E^S_{\xi,l}}}{\ln \frac{\ln 2l}{2\ln l}} &\approx \alpha.
\end{aligned}$$

Aus der Fehlerschranke $(l^{-1})^\alpha = l^{-\alpha}$ für BAKHVALOV-GITTER ergibt sich

$$\begin{aligned}
\frac{(2l)^{-\alpha}}{E^B_{\xi,2l}} &\approx \frac{l^{-\alpha}}{E^B_{\xi,l}} \\
\frac{E^B_{\xi,l}}{E^B_{\xi,2l}} &\approx \frac{(2l)^\alpha}{l^\alpha} \\
\ln \frac{E^B_{\xi,l}}{E^B_{\xi,2l}} &\approx \alpha \ln 2 \\
\frac{\ln \frac{E^B_{\xi,l}}{E^B_{\xi,2l}}}{\ln 2} &\approx \alpha.
\end{aligned}$$

In den Tabellen geben wir deshalb bei der Verwendung von SHISHKIN-Gittern an

$$\alpha^S_{2l} := \frac{\ln \frac{E^S_{\xi,2l}}{E^S_{\xi,l}}}{\ln \frac{\ln 2l}{2\ln l}}, \tag{12}$$

und bei Verwendung von BAKHVALOV-Gittern

$$\alpha^B_{2l} := \frac{\ln \frac{E^B_{\xi,l}}{E^B_{\xi,2l}}}{\ln 2} \approx \alpha. \tag{13}$$

Wir betrachten zwei Beispiele aus [11] und stellen jeweils Fehler und experimentelle Konvergenzordnungen bei Näherung der Lösungen durch quadratische, kubische und quintische Splines dar. Wir untersuchen dabei die durch unser Verfahren für das Reaktionsdiffusionsproblem

Beispiel 5.2.1.

$$-\varepsilon^2 y'' + (1 + x^2 + \cos x)y = x^{9/2} + \sin x, \quad y(0) = y(1) = 0,$$

Tabelle 5: Fehler $E^S_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.1 auf einem SHISHKIN-Gitter mit einem quadratischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}
2^7	1,30778804e-03		1,30774516e-03		1,30774515e-03	
2^8	4,16900675e-04	2,042910	4,16886151e-04	2,042913	4,16886150e-04	2,042913
2^9	1,31436959e-04	2,006244	1,31432294e-04	2,006245	1,31432293e-04	2,006245
2^{10}	4,05553242e-05	2,000488	4,05538767e-05	2,000489	4,05538765e-05	2,000489
2^{11}	1,22677460e-05	2,000033	1,22673074e-05	2,000033	1,22673074e-05	2,000033
2^{12}	3,64903824e-06	2,000393	3,64890769e-06	2,000393	3,64890768e-06	2,000393

Tabelle 6: Fehler $E^S_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.1 auf einem SHISHKIN-Gitter mit einem kubischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}
2^7	3,42730693e-06		3,42784291e-06		3,42784297e-06	
2^8	3,59313544e-07	4,030148	3,59372253e-07	4,030136	3,59372259e-07	4,030136
2^9	3,59216368e-08	4,002432	3,59274814e-08	4,002433	3,59274820e-08	4,002433
2^{10}	3,42347985e-09	3,999210	3,42404397e-09	3,999206	3,42404402e-09	3,999206
2^{11}	3,13334361e-10	3,999655	3,13385876e-10	3,999656	3,13385881e-10	3,999656
2^{12}	2,77312168e-11	4,000279	2,77357827e-11	4,000278	2,77357831e-11	4,000278

auf einem SHISHKIN-Gitter und für das Konvektionsdiffusionsproblem

Beispiel 5.2.2.

$$-\varepsilon y'' - y' + 2y = e^{x-1}, \quad y(0) = y(1) = 0,$$

sowohl auf einem SHISHKIN-, als auch auf einem BAKHVALOV-Gitter ermittelten Näherungslösungen. In den Tabellen 5, 6 und 7 können wir die erwartete gleichmäßige Konvergenz bezüglich ε für Beispiel 5.2.1 eines Reaktionsdiffusionsproblems erkennen. Die Tabellenwerte haben wir jeweils bis zu einer Genauigkeit abgebildet, welche Abweichungen zwischen den Fehlern und den experimentellen Konvergenzordnungen gerade noch erkennen lässt. Die Gitterparameter haben

Tabelle 7: Fehler $E^S_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.1 auf einem SHISHKIN-Gitter mit einem quintischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}
2^7	1,76880e-07		1,76941e-07		1,76941e-07	
2^8	2,41681e-09	7,671380242	2,41792e-09	7,671178405	2,41792e-09	7,671178385
2^9	2,30583e-11	8,085617445	2,30707e-11	8,085483647	2,30707e-11	8,085483634
2^{10}	2,12506e-13	7,973662412	2,12624e-13	7,973631829	2,12624e-13	7,973631826
2^{11}	1,77297e-15	8,006052139	1,77396e-15	8,006043860	1,77396e-15	8,006043859
2^{12}	1,38895e-17	8,000317089	1,38974e-17	8,000306540	1,38974e-17	8,000306538

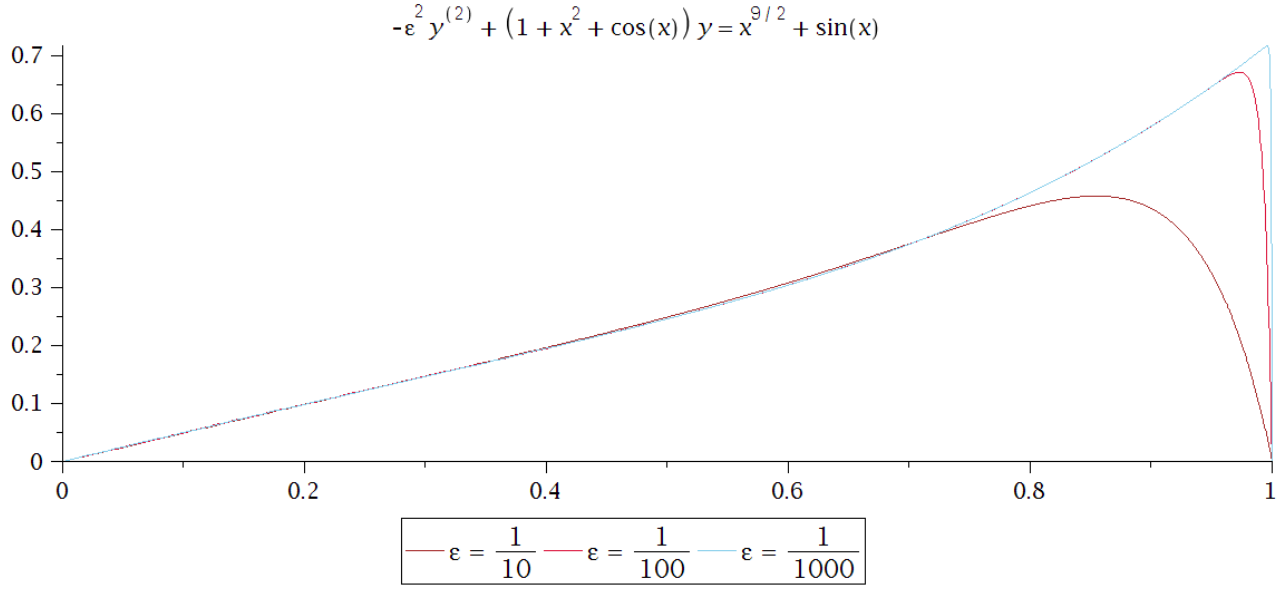

Abbildung 3: Die Graphen der Näherungslösungen von Beispiel 5.2.1 für $\varepsilon = 10^{-1}, 10^{-2}$ und 10^{-3}.

wir dabei wie folgt gewählt

$$q_0 = q_1 = \frac{1}{4},$$
$$\sigma_0 = \sigma_1 = \begin{cases} 2 & , \text{ für } k = 1, \\ 4 & , \text{ für } k = 2, \\ 8 & , \text{ für } k = 4, \end{cases}.$$

Abbildung 3 enthält die Graphen der Näherungslösungen von Beispiel 5.2.1 für $k = 1, l = 2^{10}$ und $\varepsilon = 10^{-1}, 10^{-2}, 10^{-3}$. Für unser Beispiel 5.2.2 eines Konvektionsdiffusionsproblems geben wir in den Tabellen 8, 9 und 10 Fehler und experimentelle Konvergenzordnungen auf einem SHISHKIN-Gitter und in den Tabellen 11, 12 und 13 auf einem BAKHVALOV-Gitter an. Für die Gitterparameter haben wir einheitlich $q = \frac{1}{2}$ und ebenfalls

$$\sigma = \begin{cases} 2 & , \text{ für } k = 1, \\ 4 & , \text{ für } k = 2, \\ 8 & , \text{ für } k = 4, \end{cases}$$

gewählt. Wir erkennen auch für das Konvektionsdiffusionsproblem 5.2.2 die gleichmäßige Konvergenz unseres Verfahrens auf beiden getesteten Gittern. In Abbildung 4 sind für dieses Beispiel die Graphen der Näherungslösungen für $k = 1, l = 2^{10}$ und $\varepsilon = 2^{-4}, 2^{-6}$ und $\varepsilon = 2^{-8}$ dargestellt.

6 Details zur Implementierung

Wir stellen knapp die Besonderheiten der für diese Arbeit erstellten Implementierung dar. Ein Teil der erstellen Java-Klassen findet sich im Anhang dieser Arbeit. Alle Klassen befinden sich auf dem Datenträger zu dieser Arbeit. Die Quelltexte sind zusätzlich zu den Ausführungen in

Tabelle 8: Fehler $E^S_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.2 auf einem SHISHKIN-Gitter mit einem quadratischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}
2^7	2,61101236e-3		2,61112350e-3		2,61112347e-3	
2^8	8,47926642e-4	2,009770278	8,47941203e-4	2,009815650	8,47941204e-4	2,009815631
2^9	2,66118965e-4	2,014116952	2,66121282e-4	2,014131670	2,66121282e-4	2,014131671
2^{10}	8,19128837e-5	2,004617031	8,19137910e-5	2,004612995	8,19137911e-5	2,004612995
2^{11}	2,47655656e-5	2,000883324	2,47660617e-5	2,000868343	2,47660617e-5	2,000868342
2^{12}	7,36765541e-6	2,000137043	7,36780535e-6	2,000136517	7,36780537e-6	2,000136517

Tabelle 9: Fehler $E^S_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.2 auf einem SHISHKIN-Gitter mit einem kubischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}
2^7	1,61052e-05		1,61005232e-05		1,61005440e-05	
2^8	1,72498e-06	3,9918920	1,72442359e-06	3,99195122	1,72442356e-06	3,99195355
2^9	1,71931e-07	4,0076757	1,71873911e-07	4,00770019	1,71873907e-07	4,00770020
2^{10}	1,63526e-08	4,0026564	1,63471224e-08	4,00265843	1,63471224e-08	4,00265840
2^{11}	1,49599e-09	4,0004151	1,49550268e-09	4,00040393	1,49550276e-09	4,00040384
2^{12}	1,32411e-10	4,0001537	1,32367138e-10	4,00015681	1,32367164e-10	4,00015658

Tabelle 10: Fehler $E^S_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.2 auf einem SHISHKIN-Gitter mit einem quintischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}	$E^S_{\xi,l}$	α^S_{2l}
2^7	1,6169109e-08		1,6151691e-08		1,6151689e-08	
2^8	1,7164950e-10	8,122364164	1,7144835e-10	8,122533501	1,7144833e-10	8,122533516
2^9	1,7852718e-12	7,935642192	1,7833550e-12	7,935471246	1,7833548e-12	7,935471230
2^{10}	1,6308097e-14	7,988721855	1,6290322e-14	7,988749633	1,6290320e-14	7,988749632
2^{11}	1,3671826e-16	7,997990666	1,3656573e-16	7,998033613	1,3656572e-16	7,998033598
2^{12}	1,0718135e-18	7,999148290	1,0706225e-18	7,999141097	1,0706224e-18	7,999140986

Tabelle 11: Fehler $E^B_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.2 auf einem Bakhvalov-Gitter mit einem quadratischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^B_{\xi,l}$	α^B_{2l}	$E^B_{\xi,l}$	α^B_{2l}	$E^B_{\xi,l}$	α^B_{2l}
2^7	1,1955676e-4		1,1946969e-4		1,1766158e-4	
2^8	2,9877686e-5	2,000555421	2,9855690e-5	2,000566784	2,9853865e-5	1,978653693
2^9	7,4666603e-6	2,000533405	7,4612506e-6	2,000516565	7,4612475e-6	2,000428934
2^{10}	1,8667021e-6	1,999971356	1,8653845e-6	1,999944456	1,8653841e-6	1,999944112
2^{11}	4,6666142e-7	2,000043630	4,6633564e-7	2,000032415	4,6633556e-7	2,000032413
2^{12}	1,1666462e-7	2,000009117	1,1658325e-7	2,000008104	1,1658323e-7	2,000008104

Tabelle 12: Fehler $E^B_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.2 auf einem Bakhvalov-Gitter mit einem kubischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^B_{\xi,l}$	α^B_{2l}	$E^B_{\xi,l}$	α^B_{2l}	$E^B_{\xi,l}$	α^B_{2l}
2^7	5,310000e-07		5,309599e-07		5,309600e-07	
2^8	3,327236e-08	3,996316	3,327502e-08	3,996092	3,327503e-08	3,996091
2^9	2,077951e-09	4,001091	2,078206e-09	4,001029	2,078208e-09	4,001028
2^{10}	1,298472e-10	4,000275	1,298643e-10	4,000262	1,298649e-10	4,000257
2^{11}	8,115872e-12	3,999924	8,116807e-12	3,999948	8,116949e-12	3,999930
2^{12}	5,072360e-13	4,000017	5,072712e-13	4,000083	5,073037e-13	4,000016

Tabelle 13: Fehler $E^B_{\xi,l}$ und experimentelle Konvergenzordnungen von Beispiel 5.2.2 auf einem Bakhvalov-Gitter mit einem quintischen C^1-Spline.

	ε					
	10^{-4}		10^{-8}		10^{-12}	
l	$E^B_{\xi,l}$	α^B_{2l}	$E^B_{\xi,l}$	α^B_{2l}	$E^B_{\xi,l}$	α^B_{2l}
2^7	1,5093778e-11		1,5077274e-11		1,5077272e-11	
2^8	5,8772570e-14	8,004595	5,8714233e-14	8,004450	5,8714237e-14	8,004449
2^9	2,3049360e-16	7,994272	2,3029656e-16	7,994074	2,3029713e-16	7,994070
2^{10}	8,9952675e-19	8,001345	8,9927385e-19	8,000517	8,9931042e-19	8,000462
2^{11}	3,5074299e-21	8,002608	3,5105642e-21	8,000914	3,5126502e-21	8,000115
2^{12}	1,3663398e-23	8,003954	1,3626609e-23	8,009133	1,3721001e-23	8,000030

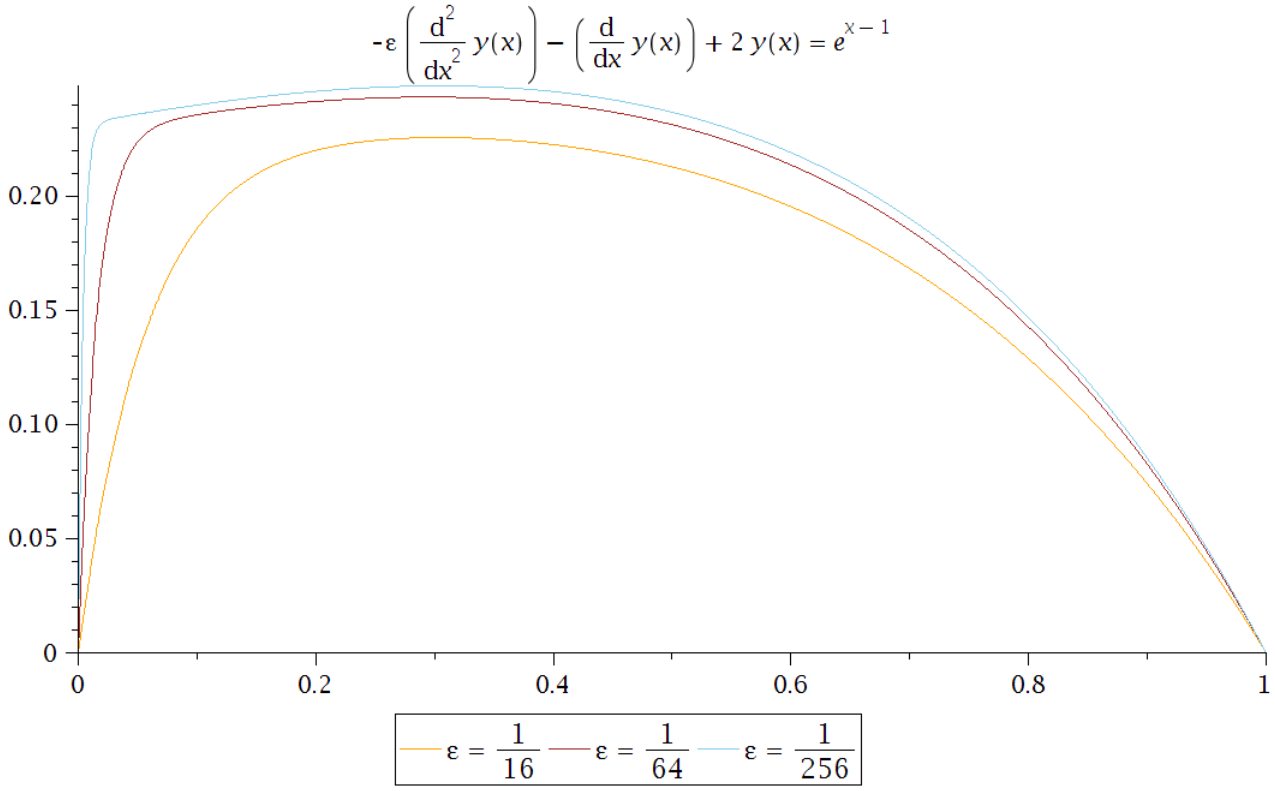

Abbildung 4: Die Graphen der Näherungslösungen von Beispiel 5.2.2 für $\varepsilon = 2^{-4}, 2^{-6}$ und 2^{-8}.

diesem Abschnitt ausführlich dokumentiert. Alle unsere Berechnungen finden mit Gleitkommazahlen bis zu einer Genauigkeit von 10^{-64} statt, so dass unsere Beispieldaten keinen relevanten Rundungsfehlern unterliegen, können aber mit beliebiger Genauigkeit von $10^{-4k}, k = 0, 1, \ldots$, durchgeführt werden. An allen Stellen, an denen Ausdrücke wiederholt benötigt werden, wie bei der Berechnungen der Stellen $\mu_0, \ldots, \mu_{kl}$ oder der Binomialkoeffizienten, werden die Werte bei der ersten Verwendung berechnet und für spätere Zugriffe gespeichert. Die Kosten für die Berechnungen wurden vor allem bezüglich der erforderlichen arithmetischen Operationen reduziert. Zu diesem Zweck wurden die Kernbestandteile der Implementierung explizit für unser Problem geschrieben und nur für den verwendeten Zahlendatentyp und periphere Aufgaben Bestandteile externer Bibliotheken verwendet. Das eigentliche Programm ist in der Klasse BezierKollokation.java zu finden, welche auch eine Funktion `berechneKlassisch` enthält, welche für den klassischen Fall verschiedene Daten auf der Konsole ausgibt, wie Maple-Plot-Befehle oder asymptotische Fehlerentwicklungen. Der singulär gestörte Fall wird in der Klasse SingulaerGestoerterFall.java behandelt, welche die Methoden `berechneReaktion` und `berechneKonvektion` zur Ausgabe der Daten enthält. Aufgrund der objektorientierten Architektur der Implementierung müssen stets jeweils ausschließlich die `main`-Methoden zur Änderung der ausgegebenen Daten verändert werden. Um andere Randwertprobleme zu behandeln, müssen ausschließlich in den genannten drei Methoden die Koeffizientenfunktionen der betrachteten Differentialgleichung, das betrachtete Intervall und die Randwerte geändert werden. Alle Implementierungsteile sind deshalb einfach in anderen Projekten wiederzuverwenden.

6.1 Binomialkoeffizienten

Die Berechnungen nutzen die Symmetrie des Binomialkoeffizienten zur Verringerung des Aufwands und sind wesentlich schneller als einige der verfügbaren Implementierungen, wenn häufig auf die Werte zugegriffen werden muss. Die Implementierung findet sich in der Klasse Binomialkoeffizient.java.

6.2 Bestimmung der Nullstellen des k-ten Legendre-Polynoms

Die den Kollokationspunkten zugrundeliegenden $(\rho_j)_{j=1}^k$ bestimmen wir mittels Satz 2.3.7 als Eigenwerte der zugeordneten JACOBI-Matrix aus Bemerkung 2.3.8. Der verwendete Algorithmus ist eine Spezialisierung des Algorithmus `imtql1` aus [14], der sich mit symmetrischen Tridiagonalmatrizen beschäftigt und Stellenauslöschung bei der Berechnung kleiner Eigenwerte reduziert. Die spezielle Eigenschaft der Nullen auf der Hauptdiagonalen unserer JACOBI-Matrix ist ebenfalls bei der Spezialisierung eingeflossen. Im Wesentlichen wurde die Implementierung der Klasse `EigenDecomposition` aus der Programmbibliothek `org.apache.commons.math3` auf die Verwendung mit dem Datentyp `Dfp` angepasst und auf das Minimum an benötigten Rechenschritten und erforderlicher Speicherauslastung für die bei unserem Problem auftretende symmetrische Tridiagonalmatrix, deren Hauptdiagonale ausschließlich Nullen enthält, angepasst. Das Ergebnis findet sich in der Klasse FieldEigenDecomposition.java.

6.3 Bernstein-Polynome

Die Bernsteinpolynome und deren Eigenschaften sind prägend in die Implementierung der Klasse BezierFunction.java eingeflossen. Dort ist die Implementierung des Algorithmus 2.1.9 von DE CASTELJAU zu finden, welcher bei der Bestimmung von Funktions- und Ableitungswerten exakt in der Form zum Einsatz kommt, wie wir das in Satz 2.1.10 angeben. Lediglich die Berechnung der ersten beiden Ableitungen wurde aus Performancegründen hart codiert.

6.4 Bézier-Splines

Die verwendeten BÉZIER-Splines wurden hauptsächlich durch das Anpassen der in der verwendeten Programmbibliothek `org.apache.commons.math3` vorhandenen Klasse `PolynomialSplineFunction` an den Datentyp `Dfp` realisiert. Außerdem wurde der Definitionsbereich gemäß unserer Definition auf ganz $\mathbb{R}$ erweitert, wie in Definition 2.2.3 angegeben.

6.5 Lösen des fast blockdiagonalen, linearen Gleichungssystems

Das Herzstück der Methode ist das Lösen des linearen Gleichungssystems. Die Untersuchung von symmetrischen und asymmetrischen blockdiagonalen Gleichungssystemen wurde unter anderem bereits in [13] vorgenommen und der schon im Abschnitt 3.3.4 erwähnte Algorithmus `bandet1` zu deren Lösung vorgeschlagen. In [4] wurde eine unter Berücksichtigung der speziellen Struktur unseres Gleichungssystems effizientere Variante dieses Algorithmus angeboten. Unsere Implementierung ist im Wesentlichen die Java-Übersetzung dieser Variante aus der `SUBROUTINE CWIDTH` des Fortran-Paketes `BSPLVB` in [4], welches für die verwendete Methode im klassischen Fall auf Basis von B-Splines ausführlich vorgestellt und angewendet wird. Dabei werden die Einträge jeder Zeile der Matrix A unseres Gleichungssystems nacheinander in ein Feld der Länge einer Zeile geschrieben und dann durch eine Variable `structure` entsprechend abgearbeitet. Unsere einzige Änderung am vorgeschlagenen Algorithmus ist die Verwendung der skalierten Spalten-Pivotsuche

bezogen auf die Zeilensummennorm statt des im Original verwendeten *partial pivoting*. Wir führen dabei allerdings keinen Vergleich der beiden Strategien durch, sondern berufen uns auf die Erwähnung unserer gewählten Strategie in [12]. Die Struktur der Matrix, der Inhalt der Variablen `structure` und der Algorithmus selbst sind in der Klasse FieldBlockDecomposition.java ausführlich dokumentiert.

6.6 Bakhvalov-Gitter

Bei der Konstruktion des Bakhvalov-Gitters für ein Konvektionsdiffusionsproblem verwenden wir, wie auch schon im Abschnitt 3.4 ausgeführt, ein iteratives Verfahren, um den Übergangspunkt τ zu bestimmen. Dabei rechnen wir solange, bis der auftretende Fehler kleiner ist, als die verwendete Genauigkeit. Dann werden alle Gitterknoten innerhalb und der erste Knoten außerhalb des Intervalls $[0, \tau]$ mittels der gittererzeugenden Funktion berechnet und danach zur Reduzierung des Aufwands der Rest des uniformen Teilgitters additiv durch $\xi_i^B = \xi_{i-1}^B + h, \forall i \colon \frac{i}{l} > \tau$, bestimmt. Die Implementierung ist in der Klasse BakhvalovGitter.java zu finden.

6.7 Shishkin-Gitter

Die Konstruktion eines Shishkin-Gitters ist sowohl für Konvektionsdiffusionsprobleme, als auch für Reaktionsdiffusionsprobleme möglich. Die Berechnung der Übergangspunkte erfolgt jeweils wie in Abschnitt 3.4 angegeben und die der Gitterknoten unter Ausnutzung der uniformen Struktur der Teilgitter analog zum uniformen Teil des Bakhvalov-Gitters. Es werden unterschiedlich breite Grenzschichten also $q_0 \neq q_1$ und $\sigma_0 \neq \sigma_1$ unterstützt. Die Implementierung findet sich in ShishkinGitter.java.

7 Zusammenfassung und Ausblick

Wir haben unter Verwendung der Bernstein-Basis eine Kollokationsmethode zur näherungsweisen Bestimmung der Lösung klassischer Zwei-Punkt Randwertprobleme auf singulär gestörte Probleme übertragen und die zu erwartenden Eigenschaften des Verfahrens in ausführlichen Tests beispielhaft beobachten können. Im Experiment erweist sich die Methode auf Shishkin-Gittern für Reaktionsdiffusionsprobleme als Verfahren vierter Ordnung und auf Shishkin- und Bakhvalov-Gittern für Konvektionsdiffusionsprobleme als Verfahren zweiter Ordnung. Die Eigenschaften der Bernstein-Basis haben dabei wesentlichen Einfluss auf die Gestalt des zu lösenden, linearen Gleichungssystems genommen. Verschiedene algorithmische Aspekte zur Bestimmung von Eigenwerten symmetrischer, tridiagonaler Matrizen und das effiziente Lösen fast blockdiagonaler Gleichungssysteme haben wir dabei ebenfalls zur Verringerung von Rechenzeiten und Rundungsfehlern behandelt, wobei wir keine ausführliche Analyse beider Aspekte durchführen.

A Die Java-Klassen

Im Folgenden finden sich einige der für diese Arbeit erstellten Java-Klassen der Implementierung des vorgestellten Verfahrens. Eine ausführliche Dokumentation ist Bestandteil des Codes und ergänzende Hinweise sind in Abschnitt 6 zu finden, weshalb hier keine weitere Erklärung ergänzt ist. Der Datenträger zu dieser Arbeit enthält den gesamten Quelltext aller verwendeten Klassen.

A.1 BezierKollokation.java

```
import org.apache.commons.lang3.ArrayUtils;
import org.apache.commons.math3.analysis.RealFieldUnivariateFunction;
import org.apache.commons.math3.dfp.Dfp;
import org.apache.commons.math3.dfp.DfpField;
import org.apache.commons.math3.util.FastMath;

/**
 * Implementierung der in der Ausarbeitung beschriebenen Kollokationsmethode
 * unter Verwendung der Bernsteinbasis.
 */
public class BezierKollokation {
    /** Die Genauigkeit, mit der gerechnet werden soll. */
    private static final int genauigkeit = 64;
    /** Der verwendete Körper in dem gerechnet wird. */
    private DfpField koerper = new DfpField(genauigkeit);
    /** Die Anzahl der Kollokationspunkte je Gitterintervall. */
    private final int k;
    /** Die Kollokationspunkte \tau_1,...,\tau_{l\cdot k}. */
    private final Dfp[] tau;
    /** Die Gitterknoten s=\xi_0,\xi_1,...,\xi_{l-1},\xi_l=t. */
    private final Dfp[] xi;
    /**
     * Die häufig auftretenden Faktoren \mu_j^r := ((\tau_j - s)/(t-s))^r,
     * j=1,...,k,  r=1,...,k+1. Es ist mu[j-1][r-1] = \mu_j^r.
     */
    private final Dfp[][] mu;
    /** Die Näherungslösung g von -\varepsilon y'' - py' + qy = f. */
    private BezierSplineFunction g;
    /** Auszug aus den möglichen Farbnamen in Maple. */
    enum Linienfarbe {
        Brown, Crimson, Chocolate, Orange, SkyBlue, Magenta, Gold
    }

    /**
     * Erzeugt eine Instanz des Näherungsverfahrens und berechnet die
     * Näherungslösung, wenn schon ein Gitter erzeugt ist.
     * @param k Anzahl der Kollokationspunkte je Gitterintervall.
     * @param xi das Gitter auf dem die Näherungslösung berechnet werden soll.
     * @param epsilon singulärer Störungsparameter.
     * @param eta1 linker Randwert.
     * @param eta2 rechter Randwert.
     * @param p Koeffizientenfunktion in -\varepsilon y'' - py' + qy = f.
     * @param q Koeffizientenfunktion in -\varepsilon y'' - py' + qy = f.
     * @param f in -\varepsilon y'' - py' + qy = f.
     */
    public BezierKollokation(int k, Gitter<Dfp> xi, Dfp epsilon,
            Dfp eta1, Dfp eta2, RealFieldPolynomialFunction p,
            RealFieldUnivariateFunction<Dfp> q,
            RealFieldUnivariateFunction<Dfp> f) {
```

```
        this.k = k;
        this.xi = xi.getXi();
        int l = this.xi.length - 1;
        tau = initialisiereTau(l);
        mu = initialisiereMus(l);
        Dfp[][] a = initialisiereA(l, epsilon, p, q);
        Dfp[] v = initialisiereV(l, eta1, eta2, f);
        Dfp[] loesung = new FieldBlockDecomposition(a, k, l)
                .solve(v);
        BezierFunction[] functions = new BezierFunction[l];
        for (int i = 0; i < l; i++) {
            functions[i] = new BezierFunction(koerper, ArrayUtils
                    .subarray(loesung, i * (k + 2), (i + 1) * (k + 2)),
                    getXi(i), getXi(i+1));
        }
        g = new BezierSplineFunction(getXi(), functions);
    }

    /**
     * Erzeugt eine Instanz des Näherungsverfahrens und berechnet die
     * Näherungslösung auf einem uniformen Gitter.
     * @param k Anzahl der Kollokationspunkte je Gitterintervall.
     * @param l die Anzahl der Gitterintervalle.
     * @param s linkes Intervallende.
     * @param t rechtes Intervallende.
     * @param eta1 linker Randwert.
     * @param eta2 rechter Randwert.
     * @param p Koeffizientenfunktion in -εy'' - py' + qy = f.
     * @param q Koeffizientenfunktion in -εy'' - py' + qy = f.
     * @param f in -εy'' - py' + qy = f.
     */
    public BezierKollokation(int k, int l, Dfp s, Dfp t, Dfp eta1, Dfp eta2,
            RealFieldUnivariateFunction<Dfp> p,
            RealFieldUnivariateFunction<Dfp> q,
            RealFieldUnivariateFunction<Dfp> f) {
        this.k = k;
        xi = initialisiereXi(l, s, t);
        tau = initialisiereTau(l);
        mu = initialisiereMus(l);
        Dfp[][] a = initialisiereA(l, koerper.getOne().negate(), p, q);
        Dfp[] v = initialisiereV(l, eta1, eta2, f);
        Dfp[] loesung = new FieldBlockDecomposition(a, k, l)
                .solve(v);
        BezierFunction[] functions = new BezierFunction[l];
        for (int i = 0; i < l; i++) {
            functions[i] = new BezierFunction(koerper, ArrayUtils
                    .subarray(loesung, i * (k + 2), (i + 1) * (k + 2)),
                    getXi(i), getXi(i+1));
        }
        g = new BezierSplineFunction(getXi(), functions);
    }
```

```
/**
 * Berechnet alle μ_j^r := μ(τ_j)^r und (1−μ_j)^r := (1−μ(τ_j))^r und speichert diese.
 * Vor dem Aufruf dieser Prozedur muss das Feld Dfp[] tau mit den
 * Kollokationspunkten initialisiert sein. Die Berechnung der
 * Funktionswerte ist dabei auf ein Minimum beschränkt unter Ausnutzung
 * von μ_j = 1 − μ_{k−j−1} und äquivalent 1 − μ_j = μ_{k−j−1}.
 * @param l die Anzahl der Gitterintervalle.
 * @return ein Feld Dfp[] mu mit mu[j][r] = μ_j^r.
 */
private Dfp[][] initialisiereMus (int l) {
    Dfp[][] tempMu = new Dfp[l*k][k+1];
    for (int i = 0; i < l; i++) {
        BezierFunction tempG = new BezierFunction(koerper,
                new Dfp[] {koerper.getZero(), koerper.getZero()},
                getXi(i), getXi(i+1));
        int index1 = i * k;
        for (int j = 0; j < k; j++) {
            int index2 = index1 + j;
            if (j > k - j)
                tempMu[index2][0] = koerper.getOne()
                .subtract(tempMu[index1 + k - j - 1][0]);
            else
                tempMu[index2][0] = tempG.getMu(getTau(index2 + 1));
            for (int r = 1; r <= k; r++) {
                tempMu[index2][r] = tempMu[index2][r-1]
                        .multiply(tempMu[index2][0]);
            }
        }
    }
    return tempMu;
}

/** Berechnet die streng monoton steigende Folge τ_j, j = 1,...,l·k. */
private Dfp[] initialisiereTau (int l) {
    Dfp[] temptau = new Dfp[l*k];
    GaussLegendrePunkte rhos = new GaussLegendrePunkte(k, koerper);
    for (int i = 0; i < l; i++) {
        Dfp tempPlus = getXi(i).add(getXi(i+1)), tempMinus = getXi(i+1)
                .subtract(getXi(i));
        for (int j = 0; j < k; j++) {
            temptau[(i * k) + j] = tempPlus.add(tempMinus.multiply(rhos
                    .getRho(j))).divide(koerper.getTwo());
        }
    }
    return temptau;
}

/**
 * Berechnet eine im Intervall [s,t] uniforme Gitterknotenfolge
 * ξ_i, i = 0,...,l mit ξ_0 = s, ξ_l = t.
```

```
     * @param s das linke Intervallende von [s,t].
     * @param t das rechte Intervallende von [s,t].
     * @param l die Anzahl der Gitterintervalle.
     */
    private Dfp[] initialisiereXi (int l, Dfp s, Dfp t) {
        return new UniformesGitter(l, s, t).getXi();
    }

    /**
     * Berechnet die Einträge der Koeffizientenmatrix A des zu lösenden
     * Gleichungssystems.
     * @param l die Anzahl der Gitterintervalle.
     * @param p in -εy'' - py' + qy = f.
     * @param q in -εy'' - py' + qy = f.
     * @return A.
     */
    private Dfp[][] initialisiereA (int l, Dfp epsilon,
            RealFieldUnivariateFunction<Dfp> p,
            RealFieldUnivariateFunction<Dfp> q) {
        /*
         * Für k=1 entsteht bei der blockweisen Erstellung die
         * Besonderheit, dass die Blöcke der Länge k+2=3 nicht mehr
         * die Stetigkeitsbedingungen aufnehmen können, da sich diese
         * jeweils auf vier Bézierpunkte beziehen. Deshalb wird in diesem
         * Fall die Matrix um eine Spalte erweitert. Eine detaillierte
         * Beschreibung der Blockstruktur von A ist in
         * FieldBlockDecomposition zu finden.
         */
        Dfp[][] tempA = new Dfp[l * (k + 2)][(k == 1 && l > 1) ? k + 3 : k+2];
        /* Variablen zum Zwischenspeichern wiederholt benötigter
         * Zwischenergebnisse und der auftretenden Binomialkoeffizienten.
         */
        Dfp deltaXi = koerper.getOne();
        Binomialkoeffizient binomM = new Binomialkoeffizient(k - 1),
                binomK = new Binomialkoeffizient(k),
                binomP = new Binomialkoeffizient(k + 1);
        /*
         * Befüllt die erste und letzte Zeile der Matrix A mit den
         * Randbedingungen.
         */
        tempA[0][0] = koerper.getOne();
        for (int i = 1; i < tempA[0].length; i++) {
            tempA[0][i] = koerper.getZero();
        }
        for (int i = 0; i < tempA[0].length - 1; i++) {
            tempA[tempA.length-1][i] = koerper.getZero();
        }
        tempA[tempA.length-1][tempA[0].length - 1] = koerper.getOne();
        /* Erzeugen der Blockstruktur durch verschachtelte Schleifen. */
        for (int i = 0; i < l; i++) {
            {
```

```
        /* Wiederholt benötigte Ausdrücke. */
        Dfp deltaXiMinus = deltaXi;
        deltaXi = getXi(i+1).subtract(getXi(i));
        /*
         * Befüllt die 2 Zeilen der Matrix A, welche aus den
         * Stetigkeitsbedingungen für den Übergang vom i-1-ten zum
         * i-ten Gitterintervall hervorgehen, i=1,...,l-1.
         */
        if (i > 0) {
            /* Die Stetigkeitsbedingung bezüglich der C^1-Stetigkeit. */
            tempA[i*(k+2)-1][0] = deltaXi;
            tempA[i*(k+2)-1][1] = deltaXiMinus.add(deltaXi).negate();
            tempA[i*(k+2)-1][2] = koerper.getZero();
            tempA[i*(k+2)-1][3] = deltaXiMinus;
            /* Die Stetigkeitsbedingung bezüglich der C^0-Stetigkeit. */
            tempA[i*(k+2)][0] = koerper.getZero();
            tempA[i*(k+2)][1] = koerper.getOne();
            tempA[i*(k+2)][2] = koerper.getOne().negate();
            tempA[i*(k+2)][3] = koerper.getZero();
            for (int j = 4; j < tempA[0].length; j++) {
                tempA[i*(k+2)-1][j] = koerper.getZero();
                tempA[i*(k+2)][j] = koerper.getZero();
            }
        }
    }
    /* Wiederholt benötigte Ausdrücke. */
    Dfp kPlusDivDeltaXi = deltaXi.reciprocal().multiply(k + 1),
            epsilonKPlusKDivDeltaXiSqr = kPlusDivDeltaXi
            .multiply(k).multiply(epsilon).divide(deltaXi),
            deltaXiSqr = deltaXi.pow(2);
    /*
     * Befüllt die k Zeilen der Matrix A, welche aus den
     * Kollokationsbedingungen des i-ten Gitterintervalls hervorgehen.
     */
    for (int j = 1; j <= k; j++) {
        /*
         * Berechnung der Funktionswerte p(τ_{ik+j}) und q(τ_{ik+j}) für den
         * j-ten Kollokationspunkt des i-ten Gitterintervalls.
         */
        Dfp pJ = p.value(getTau(i, j)), qJ = q.value(getTau(i, j));
        /*
         * Berechnung des Summanden bezüglich b_{i0} der j-ten
         * Kollokationsbedingung.
         */
        tempA[i*(k+2)+j][0] = getMu(i, j, k-1, true)
                .multiply(
                        pJ.multiply(kPlusDivDeltaXi)
                        .multiply(
                                getMu(i, j, 1, true)
                                )
                        .add(qJ.multiply(
```

```
                        getMu(i, j, 2, true)
                        ))
                .subtract(epsilonKPlusKDivDeltaXiSqr)
                );
/*
 * Berechnung des Summanden bezüglich b_i1 der j-ten
 * Kollokationsbedingung.
 */
tempA[i*(k+2)+j][1] = getMu(i, j, k-2, true)
        .multiply(k+1).multiply(
                epsilon.multiply(k)
                .divide(deltaXiSqr).multiply(
                        koerper.getTwo()
                        .subtract(getMu(i, j, 1, false)
                                .multiply(k+1))
                        )
                .subtract(
                        pJ.divide(deltaXi)
                        .multiply(
                                koerper.getOne()
                                .subtract(
                                        getMu(i,j,1,false)
                                        .multiply(k+1)
                                        )
                                )
                        .multiply(getMu(i, j, 1, true))
                        )
                .add(
                        qJ.multiply(getMu(i, j, 2, true))
                        .multiply(getMu(i, j, 1, false))
                        )
                );
/*
 * Berechnung des Summanden bezüglich b_iκ, κ = 2,...,k-1 der j-ten
 * Kollokationsbedingung.
 */
for (int kappa = 2; kappa < k ; kappa++) {
    tempA[i * (k + 2) + j][kappa] =
            epsilonKPlusKDivDeltaXiSqr
            .multiply(
                    koerper.getTwo().multiply(
                            binomM.getUeber(kappa-1))
                    .multiply(getMu(i, j, 1, true)
                            .multiply(getMu(i, j, 1, false)))
                    .subtract(getMu(i, j, 2, true)
                            .multiply(binomM
                                    .getUeber(kappa-2)))
                    .subtract(getMu(i, j, 2, false).multiply(
                            binomM.getUeber(kappa)))
                    )
            .multiply(getMu(i, j, k-1-kappa, true))
```

```
                .multiply(getMu(i, j, kappa - 2, false))

                .subtract(pJ.multiply(kPlusDivDeltaXi)
                        .multiply(
                                getMu(i, j, 1, false)
                                .multiply(binomP.getUeber(kappa))
                                .negate()
                                .add(binomK.getUeber(kappa-1))
                                )
                        .multiply(getMu(i, j, k-kappa, true))
                        .multiply(getMu(i, j, kappa-1, false)))

                .add(qJ.multiply(binomP.getUeber(kappa))
                        .multiply(getMu(i, j, k+1-kappa, true))
                        .multiply(getMu(i, j, kappa, false)));
        }
        /*
         * Berechnung des Summanden bezüglich b_ik der j-ten
         * Kollokationsbedingung.
         */
        tempA[i*(k+2)+j][k] = getMu(i, j, k-2, false)
                .multiply(k+1).multiply(
                        epsilon.multiply(k)
                        .divide(deltaXiSqr)
                        .multiply(
                                getMu(i, j, 1, false).multiply(k+1)
                                .subtract(k).add(koerper.getOne())
                                )
                        .subtract(
                                pJ.divide(deltaXi)
                                .multiply(
                                        getMu(i, j, 1, false)
                                        .multiply(k+1).negate()
                                        .add(k)
                                        )
                                .multiply(getMu(i, j, 1, false))
                                )
                        .add(
                                qJ.multiply(getMu(i, j, 2, false))
                                .multiply(getMu(i, j, 1, true))
                                )
                        );
        /*
         * Berechnung des Summanden bezüglich b_{i,k+1} der j-ten
         * Kollokationsbedingung.
         */
        tempA[i*(k+2)+j][k+1] = getMu(i,j,k-1, false)
                .multiply(
                        getMu(i, j, 1, false).multiply(
                                qJ
                        .multiply(
```

```
                                        getMu(i, j, 1, false)
                                        )
                                    .subtract(
                                        pJ.multiply(kPlusDivDeltaXi)
                                        )
                                    )
                                    .subtract(epsilonKPlusKDivDeltaXiSqr)
                                    );
                if (k == 1 && l >1) tempA[i*(k+2)+j][k+2] = koerper.getZero();
            }
        }
        return tempA;
    }

    /**
     * Berechnet die Einträge der rechten Seite v des zu lösenden
     * Gleichungssystems.
     * @param l die Anzahl der Gitterintervalle.
     * @param eta1 Randwert y(s) = η1.
     * @param eta2 Randwert y(t) = η2.
     * @param f in −εy'' + py' + qy = f.
     * @return v.
     */
    private Dfp[] initialisiereV (int l, Dfp eta1,
            Dfp eta2, RealFieldUnivariateFunction<Dfp> f) {
        Dfp[] tempV = new Dfp[l * (k + 2)];
        tempV[0] = eta1;
        for (int i = 0; i < l; i++) {
            int index = i * (k+2);
            if (i > 0) {
                tempV[index - 1] = koerper.getZero();
                tempV[index] = koerper.getZero();
            }
            for (int j = 1; j <= k; j++) {
                tempV[index + j] = f.value(getTau(i, j));
            }
        }
        tempV[l * (k+2) - 1] = eta2;
        return tempV;
    }

    /**
     * Gibt τ_j, j = 1,...,l·k zurück.
     * @param j für das τ_j, j = 1,...,l·k zurückgegeben werden soll.
     * @return τ_j
     */
    public Dfp getTau (int j) {
        return tau[j-1];
    }

    /**
```

```
     * Gibt τ_j, j=1,...,k aus dem i-ten Gitterintervall zurück.
     * @param i für das i-te Gitterintervall.
     * @param j für das τ_j, j=1,...,k zurückgegeben werden soll.
     * @return τ_{ik+j}
     */
    public Dfp getTau (int i, int j) {
        return getTau(i*k + j);
    }

    /**
     * Gibt μ_j^exponent oder (1-μ_j)^exponent zurück.
     * @param i das i-te Gitterintervall i=0,...,l-1.
     * @param j aus 1,...,k zur Auswahl von μ_j oder (1-μ_j).
     * @param exponent exponent ∈ {0,...,k+1}, für den μ_j^exponent oder (1-μ_j)^exponent.
     * @param invers true, falls (1-μ_j)^exponent und false, falls μ_j^exponent
     * zurückgegeben werden soll.
     * @return μ_j^exponent oder (1-μ_j)^exponent
     * @throws ArrayIndexOutOfBoundsException falls exponent < 0 oder exponent > k+1.
     */
    private Dfp getMu (int i, int j, int exponent, boolean invers)
            throws ArrayIndexOutOfBoundsException {
        if (exponent < -1 || exponent > k + 1 || i < 0
                || i > xi.length + 1)
            throw new ArrayIndexOutOfBoundsException();
        if (exponent == -1)
            if (invers)
                return mu[(i+1) * k - j][0].reciprocal();
            else
                return mu[i * k + j - 1][0].reciprocal();
        else if (exponent == 0)
            return koerper.getOne();
        else if (invers) {
            return mu[(i+1) * k - j][exponent-1];
        }
        else {
            return mu[i * k + j - 1][exponent-1];
        }
    }

    /**
     * Gibt die bestimmte Näherungslösung zurück.
     * @return g
     */
    public BezierSplineFunction getG () {
        return g;
    }

    /**
     * Gibt eine Kopie des Feldes Dfp[] xi der Gitterknoten
     * im Intervall [s,t] zurück.
```

```
     * @return eine Kopie von Dfp[] xi
     */
    public Dfp[] getXi() {
        Dfp out[] = new Dfp[xi.length];
        System.arraycopy(xi, 0, out, 0, xi.length);
        return out;
    }

    /**
     * Gibt ξ_i, i = 0,...,l zurück.
     * @param i für das ξ_i, i = 0,...,l zurückgegeben werden soll.
     * @return ξ_i
     */
    public Dfp getXi (int i) {
        return xi[i];
    }

    /**
     * Gibt die verwendete Genauigkeit zurück.
     * @return die Anzahl der Nachkommastellen.
     */
    public static int getGenauigkeit () {
        return new Integer(genauigkeit);
    }

    /**
     * Erzeugt für ein übergebenes k und ein l eine Instanz des
     * Kollokationsverfahrens für den klassischen Fall.
     * @param k die Anzahl der Kollokationspunkte je Gitterintervall.
     * @param l Anzahl der Gitterintervalle.
     * @param mode Gibt an, welche Testdaten auf der Konsole ausgegeben werden
     * sollen:
     * mode = 1: Gibt die Differenzen der Ableitungswerte an einigen
     * Stellen des Intervalls [0,1] aus.
     * mode = 2: Gibt die Maplebefehle zum Plotten der Näherungslösung aus.
     * mode = 3: Gibt die Abweichungen von den Kollokations- und
     * Randbedingungen aus.
     * mode = 4: Gibt die μ_j^i für alle zulässigen Parameterwerte aus.
     * mode = 5: Gibt alle τ_j aus.
     * mode = 6: Gibt alle Abweichungen der Ableitungen i-ter
     * Ordnung, i = 0,1,2, der korrespondierenden exakten Lösung und der
     * Näherungslösung aus.
     * mode = 7: Gibt alle ξ_j aus.
     * mode = 8: Berechnet die Fehler und die experimentelle
     * Konvergenzordnung für das klassische Beispiel mit l = 2^7,...,2^12.
     * Ein Aufruf in einer Schleife ist nicht nötig.
     * @return Ausgabe gemäß mode.
     */
    public static String berechneKlassisch (int k, int l, int mode) {
        String ausgabe = "k = " + k + ", l = " + l + "\n";
        /* Der Körper auf dem die Funktionen definiert sind. */
```

```
final DfpField koerper = new DfpField(genauigkeit);
/* Der linke Rand s des Kollokationsintervalls [s,t]. */
final Dfp s = koerper.getZero();
/* Der rechte Rand t des Kollokationsintervalls [s,t]. */
final Dfp t = koerper.getOne();
/* Der Randwert y(s) = η1. */
final Dfp eta1 = koerper.getZero();
/* Der Randwert y(t) = η2. */
final Dfp eta2 = koerper.getZero();
/* Die Koeffizientenfunktion p in -εy'' - py' + qy = f. */
final RealFieldPolynomialFunction p = new RealFieldPolynomialFunction(
        new Dfp[] {koerper.getZero()});
/* Die Koeffizientenfunktion q in -εy'' - py' + qy = f. */
final RealFieldPolynomialFunction q = new RealFieldPolynomialFunction(
        new Dfp[] {koerper.getTwo().multiply(koerper.getTwo())
                .negate()});
/* Die Funktion f in -εy'' - py' + qy = f. */
final RealFieldUnivariateFunction<Dfp> f = new
        RealFieldUnivariateFunction<Dfp>() {

    private final Dfp value = (koerper.getE().add(koerper.getE()
            .reciprocal())).multiply(koerper.getTwo());

    public Dfp value(Dfp x) {
        return value;
    }
};
BezierKollokation bkol = new BezierKollokation(k, l, s, t, eta1,
        eta2, p, q, f);
BezierSplineFunction g = bkol.getG();
final KlassischesBeispiel u;
Dfp x, tMinusSDivN;
int n;
switch (mode) {
case 1:
    n = 50;
    u = new KlassischesBeispiel(koerper);
    double xD;
    final double sD = s.toDouble();
    for (int i = 0; i <= n; i++) {
        xD = sD + i * (t.toDouble() - sD)/n;
        x = koerper.newDfp(xD);
        for (int j : new int[] {0, 1, 2}) {
            ausgabe += "u^(" + j + ")(" + x + ") - g^(" + j + ")(" +
                    x + ") = " + (u.getAbleitung(xD, j) -
                            g.derivative(x, j).toDouble());
            if (j < 2) ausgabe += "\n";
        }
        if (i < n) ausgabe += "\n";
    }
    break;
```

```
case 2:
    n = 200;
    String werte = "", stellen = "";
    tMinusSDivN = (t.subtract(s)).divide(n);
    for (int i = 0; i <= n; i++) {
        x = s.add(tMinusSDivN.multiply(i));
        werte += g.value(x).toDouble();
        stellen += x.toDouble();
        if (i != n) {
            werte += ",";
            stellen += ",";
        }
    }
    ausgabe = "u" + k + l + ":=pointplot([" + stellen + "],["+werte+
            "],legend=\"u" + k + l + "\",color=\"" +
            Linienfarbe.values()[(l+k-1)%Linienfarbe.values().length]
                    + "\",connect=true):";
    break;
case 3:
    ausgabe += "g(" + s + ")=" + g.value(s) + "\n";
    for (int j = 1; j <= l * k; j++) {
        ausgabe += "tau" + j + "=" + bkol.getTau(j) +
                ":g''+p*g'+q*g-f=" +
                (g.derivative(bkol.getTau(j), 2).subtract
                        (p.value(bkol.getTau(j)).multiply
                        (g.derivative(bkol.getTau(j), 1))).add
                        (q.value(bkol.getTau(j)).multiply
                        (g.derivative(bkol.getTau(j), 0))).subtract
                        (f.value(bkol.getTau(j))))
                + "\n";
    }
    ausgabe += "g(" + t + ")=" + g.value(t);
    break;
case 4:
    for (int j = 1; j <= l * k; j++) {
        for (int i = 0; i <= k+1; i++) {
            ausgabe += "mu_" + j + "^" + i + "=" +
        bkol.getMu(j/l, j%l, i, false) + "\n";
        }
    }
    break;
case 5:
    for (int i = 0; i < l; i++) {
        for (int j = 1; j <= k; j++) {
            ausgabe += "tau_" + (i * k + j) + "=" + bkol.getTau(i,
                    j) + "\n";
        }
    }
    break;
case 6:
    u = new KlassischesBeispiel(koerper);
```

```
        Dfp max = koerper.getZero();
        for (int j = 1; j <= l * k; j++) {
            Dfp temp = g.value(bkol.getTau(j)).subtract(
                    u.value(bkol.getTau(j))).abs();
            if (temp.greaterThan(max)) max = temp;
        }
        ausgabe += "E_\\Delta^" + k + (Integer.toString(k).length() == 1 ?
                "␣" : "") + "␣=␣" + max + "\n";
        max = koerper.getZero();
        for (int i = 0; i <= l; i++) {
            Dfp temp = g.value(bkol.getXi(i)).subtract(
                    u.value(bkol.getXi(i))).abs();
            if (temp.greaterThan(max)) max = temp;
        }
        ausgabe += "E_\\xi^" + l;
        for (int m = -4; m < Integer.toString(k).length() - Integer
                .toString(l).length() - (Integer.toString(k)
                        .length() == 1 ? 0 : 1); m++) {
            ausgabe += "␣";
            }
        ausgabe += "␣=␣" + max + "\n";
        break;
    case 7:
        Dfp[] tempXi = bkol.getXi();
        for (int j = 0; j <= l; j++) {
            ausgabe += "xi_" + (j + 1) + "␣=␣" + tempXi[j] + "\n";
        }
        break;
    case 8:
        ausgabe = "";
        int[] ls = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 16, 32};
        /*
         * max enthält die Fehler und experimentellen Konvergenzordnungen,
         * wobei max[0] = E_∞, max[1] = α_∞, max[2] = E_ξ, max[3] = α_ξ,
         */
        Dfp[][] konvergenz = new Dfp[ls.length][4];
        for (int i = 0; i < ls.length; i++) {
            konvergenz[i][0] = koerper.getZero();
            konvergenz[i][2] = koerper.getZero();
        }
        /*
         * Anzahl der pro Gitterintervall gleichverteilten Stellen, für
         * welche die Näherung der Maximumnorm berechnet wird. Für
         * Berechnungen die bis k = 10, l = 32 Änderungen der Norm
         * kleiner 1^-30 ergeben, sollte n ≥ 200 sein.
         */
        n = 50;
        u = new KlassischesBeispiel(koerper);
        for (int nu : new int[] {0, 1, 2}) {
            System.out.println("k␣=␣" + k + ",␣nu␣=␣" + nu + ":\n");
            for (int i = 0; i < ls.length; i++) {
```

```
int tempL = ls[i];
tMinusSDivN = (t.subtract(s)).divide(n * tempL);
bkol = new BezierKollokation (k, tempL, s,
        t, eta1, eta2, p, q, f);
g = bkol.getG();
for (int j = 0; j <= n * tempL; j++) {
    x = s.add(tMinusSDivN.multiply(j));
    Dfp temp = g.derivative(x, nu).subtract(
            u.getAbleitung(x, nu)).abs();
    if (temp.greaterThan(konvergenz[i][0])) {
        konvergenz[i][0] = temp;
    }
}
System.out.print("l = ");
for (int m = 0; m < 2 - Integer.toString(ls[i]).length();
        m++) {
    System.out.print(" ");
};
System.out.println(ls[i] + ": E_\\infty = " +
konvergenz[i][0]);
if (i > 0) {
    konvergenz[i][1] = konvergenz[i][0].divide(
            konvergenz[i-1][0]).log().divide(
            FastMath.log((double)ls[i-1] / ls[i]));
    for (int m = 0; m < 2 - Integer.toString(ls[i])
            .length(); m++) {
        System.out.print(" ");
    };
    System.out.println("        \\alpha_" + ls[i] + " = " +
            konvergenz[i][1]);
}
for (int j = 0; j <= ls[i]; j++) {
    Dfp temp = g.derivative(bkol.getXi(j), nu).subtract(
            u.getAbleitung(bkol.getXi(j), nu)).abs();
    if (temp.greaterThan(konvergenz[i][2])) {
        konvergenz[i][2] = temp;
    }
}
System.out.println("           E_\\xi = " +
konvergenz[i][2]);
if (i > 0) {
    konvergenz[i][3] = konvergenz[i][2].divide(
            konvergenz[i-1][2]).log().divide(FastMath
                    .log((double)ls[i-1] / ls[i]));
    for (int m = 0; m < 2 - Integer.toString(ls[i])
            .length();
            m++) {
        System.out.print(" ");
    };
    System.out.println("        \\beta_" + ls[i] + " = " +
            konvergenz[i][3]);
```

```
                    }
                    System.out.println();
                }
            }
            break;
        }
        return ausgabe;
    }

    /**
     * Berechnen verschiedener Daten in Abhängigkeit des angegebenen Modus
     * und Ausgabe auf der Konsole.
     */
    public static void main (String[] args) {
        int[] n = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 16, 32};
        for (int k : new int[] {1, 2, 4}) {
            for (int l : n) {
                System.out.println(berechneKlassisch(k, l, 6));
            }
        }
    }
}
```

A.2 BezierFunction.java

```
import org.apache.commons.math3.analysis.RealFieldUnivariateFunction;
import org.apache.commons.math3.dfp.Dfp;
import org.apache.commons.math3.dfp.DfpField;
import org.apache.commons.math3.exception.NoDataException;
import org.apache.commons.math3.exception.NullArgumentException;
import org.apache.commons.math3.exception.util.LocalizedFormats;
import org.apache.commons.math3.util.MathUtils;

/**
 * Repräsentiert ein Polynom in Bézierdarstellung angelehnt an
 * org.apache.commons.math3.analysis.polynomials.PolynomialFunction.
 */
public class BezierFunction implements RealFieldUnivariateFunction<Dfp> {

    /** Der Körper auf dem die Bézierdarstellung implementiert wird. */
    DfpField koerper;
    /**
     * The bezierpoints of the polynomial, ordered by degree -- i.e.,
     * b[0] is b_0 and b[n] is the coefficient of {}_sB_n^t where n is the
     * degree of all Bernstein polynomials the function is composed of.
     */
    private final Dfp[] b;
    /**
     * Die Intervallgrenzen [s,t] über dem die Basispolynome
     * definiert sind und häufig auftretende Rechenschritte.
     */
    private final Dfp tMinusS, tMinusSSqr, SDivTMinusS;
```

```
/**
 * Construct a polynomial with the given bezierpoints. The first
 * element of Dfp[] b is the bezierpoint for ${}_s^tB_0^{k+1}$.
 * Higher degree bezierpoints follow in sequence. The degree of the
 * resulting polynomial is the index of the last element of the array.
 * The constructor makes a copy of the input array and assigns the
 * copy to Dfp[] b.
 * @param b bezierpoints.
 * @throws NullArgumentException if b is null.
 * @throws NoDataException if b is empty.
 */
public BezierFunction(DfpField koerper, Dfp[] b, Dfp s, Dfp t)
        throws NullArgumentException, NoDataException {
    this.koerper = koerper;
    tMinusS = t.subtract(s);
    tMinusSSqr = tMinusS.pow(2);
    SDivTMinusS = s.divide(tMinusS);
    MathUtils.checkNotNull(b);
    int n = b.length;
    if (n == 0) {
        throw new NoDataException(
                LocalizedFormats
                .EMPTY_POLYNOMIALS_COEFFICIENTS_ARRAY);
    }
    this.b = new Dfp[n];
    System.arraycopy(b, 0, this.b, 0, n);
}

/**
 * Auswertung des Polynoms in Bézierdarstellung an der Stelle x.
 * @param x die Stelle an der ausgewertet werden soll.
 * @return Funktionswert an der Stelle x.
 */
public Dfp value (Dfp x) {
    return deCasteljau(x, getMu(x), b.length - 1, 0);
}

/**
 * Auswertung der m-ten Ableitung, m = 0,1,2,... des
 * Polynoms in Bézierdarstellung an der Stelle x.
 * @param x die Stelle an der ausgewertet werden soll.
 * @param nu Grad der Ableitung, die ausgewertet werden soll.
 * @return Ableitungsswert an der Stelle x.
 */
public Dfp derivative (Dfp x, int nu) {
    Dfp mu = getMu(x);
    if (nu == 0) return value(x);
    /*
     * Switch-Anweisung, weil die ersten drei Ableitungen
     * nu = 0,1,2 aus Geschwindigkeitsgründen hart codiert sind.
```

```
     */
    switch (nu) {
    case 0:
        return value(x);
    case 1:
        return tMinusS.reciprocal()
        .multiply(b.length - 1).multiply(deCasteljau(x, mu, b.length
                -2, 1).subtract(deCasteljau(x, mu, b.length - 2, 0)));
    case 2:
        return tMinusSSqr.reciprocal()
        .multiply(b.length - 1).multiply(b.length - 2)
        .multiply(deCasteljau(x, mu, b.length - 3, 0)
                .subtract(koerper.getTwo().multiply(
                        deCasteljau(x, mu, b.length - 3, 1)))
                .add(deCasteljau(x, mu, b.length - 3, 2)));
    default:
        Dfp tempValue = koerper.getZero();
        Binomialkoeffizient mUeberI = new Binomialkoeffizient(nu);
        int n = 1;
        for (int i = 0; i <= nu; i++) {
            tempValue = tempValue.add(((nu-i)%2 == 0) ?
                    deCasteljau(x, mu, b.length-1-nu, i)
                    .multiply(mUeberI.getUeber(i))
                    : deCasteljau(x, mu, b.length-1-nu, i)
                    .multiply(mUeberI.getUeber(i)).negate());
            if (i < nu) n *= b.length - 1 - i;
        }
        return tempValue.multiply(n).divide(tMinusS.pow(nu));
    }
}

/**
 * Führt den Algorithmus von deCasteljau durch.
 * @param x Stelle die ausgewertet werden soll.
 * @param mu Quotient (x-s)/(t-s) für die Berechnung der Rekursion
 * mit den Intervallgrenzen [s,t].
 * @param r oberer Index b_i^r.
 * @param i unterer Index b_i^r.
 * @param b Feld der Bézier-Punkte.
 * @return Für i=0,r=n der Funktionswert des Polynoms in
 * Bézier-Darstellung mit Bézier-Punkten doube[] b an der
 * Stelle x.
 */
public Dfp deCasteljau (Dfp x, Dfp mu, int r, int i) {
    if (r == 0)
        return b[i];
    else
        return  (mu.multiply(deCasteljau(x, mu, r - 1, i + 1)))
                .add(koerper.getOne().subtract(mu)
                        .multiply(deCasteljau(x, mu, r - 1, i)));
}
```

```
        /**
         * Berechnet den häufig auftretenden Faktor μ(x) := (x-s)/(t-s).
         * @param x Wert, für den μ(x) berechnet werden soll.
         * @return μ(x)
         */
        public Dfp getMu (Dfp x) {
            return x.divide(tMinusS).subtract(SDivTMinusS);
        }

        /**
         * Returns a copy of the bezierpoints. Changes made to the returned
         * copy will not affect the bezierpoints of the polynomial.
         * @return a fresh copy of the bezierpoints array.
         */
        public Dfp[] getBezierpunkte() {
            return (Dfp[]) b.clone();
        }

        /**
         * Berechnet den häufig auftretenden Faktor μ(x) := (x-s)/(t-s).
         * @param x Wert, für den μ(x) berechnet werden soll.
         * @return μ(x)
         */
        public static Dfp mu (Dfp x, Dfp s, Dfp t) {
            return (x.subtract(s)).divide(t.subtract(s));
        }
    }
```

A.3 FieldEigenDecomposition.java

```
import org.apache.commons.math3.dfp.Dfp;
import org.apache.commons.math3.dfp.DfpField;
import org.apache.commons.math3.exception.MaxCountExceededException;
import org.apache.commons.math3.exception.util.LocalizedFormats;

/**
 * Calculates the eigenvalues of a real matrix taking the algortihm from
 * EigenDecomposition and adapting it to Dfp matrices. This class is similar
 * in spirit to the EigenvalueDecomposition class from the JAMA library, with the
 * limitation to calculating Eigenvalues of symmetric tridiagonal matrices
 * with only zeros on the main diagonal to save operations. This
 * implementation is based on the paper by A. Drubrulle, R.S. Martin and
 * J.H. Wilkinson "The Implicit QL Algorithm" in Wilksinson and Reinsch
 * (1971) Handbook for automatic computation, vol. 2, Linear algebra,
 * Springer-Verlag, New-York.
 */
public class FieldEigenDecomposition {
    /** Der Körper über dem die Zerlegung berechnet werden soll. */
    private final DfpField koerper;
    /** Maximum number of iterations accepted. */
    private final static byte maxIter = 30;
```

```
/** The eigenvalues. */
private final Dfp[] eigenvalues;

/**
 * Calculates the eigen decomposition of the symmetric tridiagonal
 * matrix.
 * @param neben of the tridiagonal form.
 * @throws MaxCountExceededException if the algorithm fails to converge.
 * @since 3.1
 */
public FieldEigenDecomposition(final Dfp[] neben) {
        this.koerper = neben[0].getField();
        eigenvalues = findEigenValues(neben);
    }

/**
 * Find eigenvalues (DUBRULLE et al., 1971)
 * @param neben Secondary diagonal of the tridiagonal matrix.
 */
private Dfp[] findEigenValues(Dfp[] neben) {
    final int n = neben.length;
    Dfp[] tempEigenvalues = new Dfp[n];
    final Dfp[] e = neben;
    for (int i = 0; i < n; i++) {
        tempEigenvalues[i] = koerper.getZero();
    }

    for (int j = 0; j < n; j++) {
        int its = 0;
        int m;
        do {
            for (m = j; m < n - 1; m++) {
                Dfp delta = (tempEigenvalues[m].abs())
                        .add(tempEigenvalues[m + 1].abs());
                if ((e[m].abs()).add(delta).equals(delta)) {
                    break;
                }
            }
            if (m != j) {
                if (its == maxIter) {
                    throw new MaxCountExceededException(
                            LocalizedFormats.CONVERGENCE_FAILED, maxIter);
                }
                its++;
                Dfp q = (tempEigenvalues[j + 1]
                        .subtract(tempEigenvalues[j]))
                        .divide(e[j].multiply(2));
                Dfp t = (koerper.getOne().add(q.pow(2))).sqrt();
                if (q.lessThan(koerper.getZero())) {
                    q = tempEigenvalues[m].subtract(tempEigenvalues[j])
                            .add(e[j].divide(q.subtract(t)));
```

```
                } else {
                    q = tempEigenvalues[m].subtract(tempEigenvalues[j])
                            .add(e[j].divide(q.add(t)));
                }
                Dfp u = koerper.getZero();
                Dfp s = koerper.getOne();
                Dfp c = koerper.getOne();
                int i;
                for (i = m - 1; i >= j; i--) {
                    Dfp p = s.multiply(e[i]);
                    Dfp h = c.multiply(e[i]);
                    if ((p).abs().greaterThan(q.abs()) ||
                            (p).abs().equals(q.abs())) {
                        c = q.divide(p);
                        t = (c.pow(2).add(koerper.getOne())).sqrt();
                        e[i + 1] = p.multiply(t);
                        s = t.reciprocal();
                        c = c.multiply(s);
                    } else {
                        s = p.divide(q);
                        t = (s.pow(2).add(koerper.getOne())).sqrt();
                        e[i + 1] = q.multiply(t);
                        c = t.reciprocal();
                        s = s.multiply(c);
                    }
                    if (e[i + 1].isZero()) {
                        tempEigenvalues[i + 1] = tempEigenvalues[i + 1]
                                .subtract(u);
                        e[m] = koerper.getZero();
                        break;
                    }
                    q = tempEigenvalues[i + 1].subtract(u);
                    t = ((tempEigenvalues[i].subtract(q)).multiply(s))
                            .add(c.multiply(h).multiply(2));
                    u = s.multiply(t);
                    tempEigenvalues[i + 1] = q.add(u);
                    q = (c.multiply(t)).subtract(h);
                }
                if (t.isZero() && i >= j) {
                    continue;
                }
                tempEigenvalues[j] = tempEigenvalues[j].subtract(u);
                e[j] = q;
                e[m] = koerper.getZero();
            }
        } while (m != j);
    }
    /* Sort the eigen values in increase order. */
    for (int i = 0; i < n; i++) {
        int k = i;
        Dfp p = tempEigenvalues[i];
```

```
            for (int j = i + 1; j < n; j++) {
                if (tempEigenvalues[j].lessThan(p)) {
                    k = j;
                    p = tempEigenvalues[j];
                }
            }
            if (k != i) {
                tempEigenvalues[k] = tempEigenvalues[i];
                tempEigenvalues[i] = p;
            }
        }
        return tempEigenvalues;
    }

    /**
     * Gets a copy of the eigenvalues of the original matrix.
     * @return a copy of the eigenvalues of the original matrix.
     */
    public Dfp[] getEigenvalues() {
        return eigenvalues.clone();
    }
}
```

A.4 FieldBlockDecomposition.java

```
import org.apache.commons.math3.dfp.Dfp;
import org.apache.commons.math3.dfp.DfpField;
import org.apache.commons.math3.exception.DimensionMismatchException;
import org.apache.commons.math3.linear.*;

/**
 * Implementierung des Algorithmus bandet1 von Martin und Wilkinson (
 * Numer. Math. 9 (1967) pp. 279--301) zur direkten Lösung eines linearen
 * Gleichungssystems mit einer asymmetrischen Bandmatrix in der Version von
 * de Boor (A practical guide to splines (1978) pp. 370--372,
 * SUBROUTINE CWIDTH), der den Algorithmus noch auf die spezielle
 * fast blockdiagonale Struktur der bei Spline-Kollokationen auftretenden
 * Matrizen angepasst hat. Lediglich die skalierte Spalten-Pivotsuche nehmen
 * wir bezogen auf die Zeilensummennorm vor und nicht, wie im Original, in
 * Bezug auf die Maximumnorm, weil diese als eine von zwei Pivotstrategien,
 * welche die Restmatrix einbeziehen, in T. Linss (Numerische Mathematik I
 * (2015) pp. 77-78) genannt wird. Die Blockstruktur der Kollokationsmatrix
 * ist dabei auf eine Kollokationsmatrix zur Bestimmung eines C^1-Splines
 * ausgelegt, der eine Zwei-Punkt-Randwertaufgabe löst. Das heißt die Matrix
 * A beginnt mit einer Zeile zur Randbedingungen am Anfang des betrachteten
 * Intervalls, gefolgt von k Kollokationsbedingungen, auf die wiederum bis
 * zum vorletzten der l-1 Blöcke 2 Stetigkeitsbedingungen folgen und an
 * die sich im letzten Block die Randbedingung am rechten Ende des
 * betrachteten Intervalls anschließt.
 */
public class FieldBlockDecomposition {
```

```
/** Körper auf dem die Zerlegung definiert ist. */
private final DfpField koerper;
/** Matrix A im Gleichungssystem Ax=b. */
private final Dfp[][] a;
/** Anzahl der Blöcke in der Matrix A. */
private final int k;
/**
 * Die Anzahl der Kollokationsbedingungen je Block. */
private final int l;

/**
 * Erzeugt eine Instanz des Zerlegers der Matrix A.
 * @param a die Matrix A für die Ax=b gelöst werden soll.
 * @param b der Vektor b für den Ax=b gelöst werden soll.
 * @param k die Anzahl der Kollokationsbedingungen mit der die Matrix
 * erstellt wurde.
 * @param l die Anzahl der Gitterintervalle.
 */
public FieldBlockDecomposition (Dfp[][] a, final int k,
        final int l) {
    if (a.length != l * (k + 2) && a[0].length == k+2)
        throw new DimensionMismatchException(a.length, l * (k+2));
    if (k == 1 && l > 1) {
        if (a[0].length != 4)
            throw new DimensionMismatchException(a[0].length, k+2);
    } else {
        if (a[0].length != k+2)
            throw new DimensionMismatchException(a[0].length, k+2);
    }
    this.k = k;
    this.l = l;
    koerper = (DfpField) a[0][0].getField();
    this.a = a;
}

/**
 * Löst für einen übergebenen Vektor b das Gleichungssystem Ax=b.
 * @param b die rechte Seite des Gleichungssystems.
 * @return Die Lösung x des Gleichungssystems Ax=b.
 */
public Dfp[] solve(Dfp[] b) {
    /*
     * Folgende Variablenbezeichnungen verwenden wir im
     * Vergleich zu DE BOOR (1978):
     * W =: a;
     * B =: b;
     * NEQU =: a.length;
     * COLMAX =: maxDominanz;
     * TEMP =: maxDominanz;
     * AWILOD =: tempDominanz;
     * ISTAR =: indexMaxDominanz;
```

```
 * NCOLS =: a[0].length;
 * LASTCL =: letzteSpalteBlock;
 * NROWAD =: structure[i][0];
 * NEXTEQ =: letzteGleichungBlock + j;
 * INTEGS =: structure;
 * NBLOKS =: 2*l - 1 = structure.length -1;
 * D =: zeilenSumme;
 * X =: tempX;
 * IFLAG nicht vorhanden, weil wir keine Determinante berechnen;
 * IPVTEQ =: pivot;
 * LASTEQ =: letzteGleichungBlock;
 * PVTL =: pivot +1;
 * I =: i;
 * II =: restBlock;
 * jMax =: restBlock;
 * ICOUNT =: j;
 * RATIO =: ratio;
 * SUM =: tempSumme;
 */
if (b.length != a.length)
    throw new DimensionMismatchException(b.length, a.length);
Dfp[] tempX = new Dfp[b.length];
Dfp[] zeilenSumme = new Dfp[b.length];
/*
 * Nimmt die Struktur der Blockmatrix auf. Dabei ist structure[i][0]
 * die Anzahl der Zeilen in Block i und structure[i][1] die Anzahl
 * der möglichen Pivotschritte in Block i, bevor möglichweise eine
 * Zeile aus dem nächsten Block in den Block hineinpivotiert wird.
 */
int[][] structure;
if (l == 1) {
    /*
     * Im Falle, dass nur ein Block vorhanden ist, wird über alle
     * Zeilen pivotiert.
     */
    structure = new int[][] {{k+2, k+2}};
} else {
    /*
     * Sonst bilden die jeweils k Kollokationsbedingungen jedes
     * Gitterintervalls einen Block, am oberen und unteren Rand der
     * Matrix zusammen mit den Randbedingungen, und jeweils die
     * zwei Zeilen der Stetigkeitsbedingungen zwischen den
     * Kollokationsbedingungen.
     */
    structure = new int[2 * l - 1][2];
    structure[0] = new int[] {k+1, k};
    for (int i = 1; i < l; i++) {
        structure[2*i-1] = new int[] {2, 2};
        structure[2*i] = new int[] {k, k};
    }
    /*
```

```
         * Für k=1 entsteht bei der blockweisen Bearbeitung die
         * Besonderheit, dass die Blöcke der Länge k+2=3 nicht mehr
         * die Stetigkeitsbedingungen aufnehmen können, da sich diese
         * jeweils auf vier Bézierpunkte beziehen. Deshalb wird in diesem
         * Fall die Matrix geringfügig umsortiert und structure
         * angepasst. Der vorletzte Block besteht dann lediglich aus der
         * C^1-Stetigkeitsbedingung und die C^0_Stetigkeitsbedingung
         * rutscht in den letzten Block. Dafür muss dann die zugehörige
         * Zeile in der Matrix um eins nach links und die letzte
         * Kollokationsbedingung eins nach rechts verschoben werden.
         */
        if (k == 1) {
            structure[structure.length-2] = structure[structure.length-1];
            structure[structure.length-1] = new int[] {3, 4};
            a[(l - 1) * 3][0] = a[(l - 1) * 3][1];
            a[(l - 1) * 3][1] = a[(l - 1) * 3][2];
            a[(l - 1) * 3][2] = koerper.getZero();
            a[(l - 1) * 3 + 1][3] = a[(l - 1) * 3 + 1][2];
            a[(l - 1) * 3 + 1][2] = a[(l - 1) * 3 + 1][1];
            a[(l - 1) * 3 + 1][1] = a[(l - 1) * 3 + 1][0];
            a[(l - 1) * 3 + 1][0] = koerper.getZero();
        } else
            structure[structure.length-1] = new int[] {k+1, k+2};
    }
    int pivot = -1, letzteGleichungBlock = 0;
    {
        for (int i = 0; i < structure.length; i++) {
            {
                /*
                 * In diesem Bereich wird für die skalierte Spalten-
                 * Pivot-Suche für jeden in die Elimination erstmals
                 * eintretenden Block jeweils für alle Zeilen die Summe
                 * der Beträge ihrer Einträge berechnet.
                 */
                for (int j = 0; j < structure[i][0]; j++) {
                    Dfp tempZeilenSumme = koerper.getZero();
                    for (int m = 0; m < a[i].length; m++) {
                        tempZeilenSumme = tempZeilenSumme
                                .add(a[letzteGleichungBlock+ j][m].abs());
                    }
                    if (tempZeilenSumme == koerper.getZero())
                        throw new SingularMatrixException();
                    zeilenSumme[letzteGleichungBlock+j] = tempZeilenSumme;
                }
            }
            /* Bezieht den nächsten Block in die Betrachtung ein. */
            letzteGleichungBlock += structure[i][0];
            /*
             * Im aktuellen Block wird jeweils mit voller Spaltenanzahl
             * gestartet und pro Pivotschritt eine Spalte weniger
             * berücksichtigt. Die potentiellen Nichtnullelemente des
```

```
         * Blocks stehen dann immer beginnend mit Spalte eins in a.
         */
        int letzteSpalteBlock = a[0].length;
        for (int j = 1; j <= structure[i][1]; j++) {
            /*
             * Wandert durch den aktuellen Block bis möglicherweise
             * Zeilen aus dem nächsten Block hineinpivotiert werden.
             */
            pivot++;
            if (pivot < letzteGleichungBlock) {
                {
                    /*
                     * In diesem Bereich wird die skalierte Spalten-
                     * Pivot-Suche durchgeführt und gegebenenfalls der
                     * Austausch der Zeilen vorgenommen.
                     */
                    int indexMaxDominanz = pivot;
                    Dfp maxDominanz = a[pivot][0].abs()
                            .divide(zeilenSumme[pivot]);
                    for (int m = pivot + 1; m < letzteGleichungBlock;
                            m++) {
                        Dfp tempDominanz = a[m][0].abs()
                                .divide(zeilenSumme[m]);
                        if (tempDominanz.greaterThan(maxDominanz)) {
                            maxDominanz = tempDominanz;
                            indexMaxDominanz = m;
                        }
                    }
                    if (indexMaxDominanz != pivot) {
                        maxDominanz = zeilenSumme[indexMaxDominanz];
                        zeilenSumme[indexMaxDominanz] =
                                zeilenSumme[pivot];
                        zeilenSumme[pivot] = maxDominanz;
                        maxDominanz = b[indexMaxDominanz];
                        b[indexMaxDominanz] = b[pivot];
                        b[pivot] = maxDominanz;
                        for (int m = 0; m < letzteSpalteBlock; m++) {
                            maxDominanz = a[indexMaxDominanz][m];
                            a[indexMaxDominanz][m] = a[pivot][m];
                            a[pivot][m] = maxDominanz;
                        }
                    }
                }
                /*
                 * Durchführung der Elimination und verschieben aller
                 * verbleibenden Elemente um eine Spalte nach links.
                 */
                for (int restBlock = pivot + 1;
                        restBlock<letzteGleichungBlock; restBlock++) {
                    Dfp ratio = a[restBlock][0].divide(a[pivot][0]);
                    for (int m = 1; m < letzteSpalteBlock; m++) {
```

```
                                a[restBlock][m-1] = a[restBlock][m]
                                        .subtract(ratio
                                                .multiply(a[pivot][m]));
                                }
                                a[restBlock][letzteSpalteBlock-1] = koerper
                                        .getZero();
                                b[restBlock] = b[restBlock]
                                        .subtract(ratio.multiply(b[pivot]));
                            }
                            letzteSpalteBlock--;
                        } else if (a[pivot][0].isZero())
                            throw new SingularMatrixException();
                    }
                }
            }
            /* Initialisierung des Lösungsvektors. */
            for (int i = 0; i < tempX.length; i++) {
                tempX[i] = koerper.getZero();
            }
            /* Rückwärtsersetzung unter Berücksichtigung der Struktur von A. */
            for (int i = structure.length - 1; i >= 0; i--) {
                int restBlock = a[0].length - structure[i][1];
                for (int j = 0; j < structure[i][1]; j++) {
                    Dfp tempSumme = koerper.getZero();
                    for (int m = 1; m <= restBlock; m++) {
                        tempSumme = tempSumme.add(tempX[pivot+m]
                                .multiply(a[pivot][m]));
                    }
                    tempX[pivot]=b[pivot].subtract(tempSumme)
                            .divide(a[pivot][0]);
                    restBlock++;
                    pivot--;
                }
            }
            return tempX;
        }
}
```

A.5 ShishkinGitter.java

```
import org.apache.commons.math3.dfp.Dfp;

/**
 * Repräsentiert ein SHISHKIN-Gitter auf einem beliebigen Intervall [s,t] mit
 * l Teilintervallen in Form eines Dfp [] Feldes, welches die Knoten enthält.
 */
public class ShishkinGitter implements GrenzschichtAngepasstesGitter<Dfp>{

    /** Enthält die Knoten. */
    private final Dfp[] xiS;
    /**
     * Enthält die Überganspunkte τ:
```

```
 */
private Dfp tau, tau_0, tau_1;

/**
 * Erzeugt ein Shishkin-Gitter ξ^S = (ξ_i^S) mit l+1 Knoten für eine
 * Konvektionsdiffusionsgleichung.
 * @param l die Anzahl der Gitterintervalle.
 * @param s der erste Gitterknoten ξ_0^S.
 * @param t der letzte Gitterknoten ξ_l^S.
 * @param q ein Gitterparamter beschreibt größenordnungsmäßig den
 * Anteil der Gitterknoten, welcher in der Grenzschicht liegt.
 * @param sigma ein Gitterparamter beschreibt die Auflösung der
 * Grenzschicht und wird typischerweise nahe der formalen
 * Konvergenzordnung der verwendeten Methode gewählt.
 * @param beta gegeben durch die Koeffzienten der Differentialgleichung
 * −εu'' − bu' + cu = f, für die das Gitter erzeugt wird, durch b ≥ β > 0.
 * @param epsilon singulärer Störungsparamter für welchen das Gitter
 * erzeugt werden soll.
 */
public ShishkinGitter(int l, Dfp s, Dfp t, Dfp q, Dfp sigma, Dfp beta,
        Dfp epsilon) {
    xiS = new Dfp[l+1];
    xiS[0] = s;
    int qL = q.newInstance(q.multiply(l).floor()).intValue();
    /* Berechnung des Gitterübergangpunktes τ. */
    tau = sigma.multiply(epsilon).divide(beta).multiply(Math.log(l));
    if (q.lessThan(tau)) tau = q;
    /* Berechnung der lokal uniformen Teilgitter... */
    Dfp TMinusS = t.subtract(s);
    {
        /*... im Intervall [s, s+τ(t−s)]. */
        Dfp tauTMinusSDivQL = tau.multiply(TMinusS).divide(qL);
        for (int i = 1; i <= qL; i++) {
            xiS[i] = xiS[i-1].add(tauTMinusSDivQL);
        }
    } {
        /*... im Intervall [s+τ(t−s), t], t]. */
        Dfp einsMinustauTMinusSDivQL = tau.getOne().subtract(tau)
                .multiply(TMinusS).divide(l - qL);
        for (int i = qL + 1; i <= l; i++) {
            xiS[i] = xiS[i-1].add(einsMinustauTMinusSDivQL);
        }
    }
}

/**
 * Erzeugt ein Shishkin-Gitter ξ^S = (ξ_i^S) mit l+1 Knoten für eine
 * Reaktionsdiffusionsgleichung.
 * @param l die Anzahl der Teilintervalle.
 * @param s der erste Gitterknoten ξ_0^S.
 * @param t der letzte Gitterknoten ξ_l^S.
```

```
     * @param q_0 ein Gitterparamter beschreibt größenordnungsmäßig den
     * Anteil der Gitterknoten, welcher in der Grenzschicht bei x=s liegt.
     * @param q_1 ein Gitterparamter beschreibt größenordnungsmäßig den
     * Anteil der Gitterknoten, welcher in der Grenzschicht bei x=t liegt.
     * @param sigma_0 ein Gitterparamter beschreibt die Auflösung der
     * Grenzschicht bei x=s und wird typischerweise nahe der formalen
     * Konvergenzordnung der verwendeten Methode gewählt.
     * @param sigma_1 ein Gitterparamter beschreibt die Auflösung der
     * Grenzschicht bei x=t und wird typischerweise nahe der formalen
     * Konvergenzordnung der verwendeten Methode gewählt.
     * @param gamma gegeben durch die Koeffzienten der Differentialgleichung
     * -εu''+cu=f, für die das Gitter erzeugt wird, durch c≥γ^2,γ>0.
     * @param epsilon singulärer Störungsparamter für welchen das Gitter
     * erzeugt werden soll.
     */
    public ShishkinGitter(int l, Dfp s, Dfp t, Dfp q_0, Dfp q_1, Dfp sigma_0,
            Dfp sigma_1, Dfp gamma, Dfp epsilon) {
        xiS = new Dfp[l+1];
        xiS[0] = s;
        int q_0L = q_0.newInstance(q_0.multiply(l).floor()).intValue(),
                q_1L = q_1.newInstance(q_1.multiply(l).floor()).intValue();
        /* Berechnung der Gitterübergangpunkte τ_0 und τ_1. */
        {
            Dfp temp = epsilon.divide(gamma).multiply(Math.log(l));
            tau_0 = sigma_0.multiply(temp);
            tau_1 = sigma_1.multiply(temp);
        }
        if (q_0.lessThan(tau_0)) tau_0 = q_0;
        if (q_1.lessThan(tau_1)) tau_1 = q_1;
        /* Berechnung der lokal uniformen Teilgitter... */
        Dfp TMinusS = t.subtract(s);
        {
            /*... im Intervall [s,s+τ_0(t-s)]. */
            Dfp tau_0TMinusSDivQ_0L = tau_0.multiply(TMinusS).divide(q_0L);
            for (int i = 1; i <= q_0L; i++) {
                xiS[i] = xiS[i-1].add(tau_0TMinusSDivQ_0L);
            }
        }
        {
            /*... im Intervall [s+τ_0(t-s),t-τ_1(t-s)]. */
            Dfp einsMinustau_0Plustau1TMinusSDivQL = tau_0.getOne().subtract(
                    tau_0).subtract(tau_1).multiply(TMinusS)
                    .divide(l - (q_0L + q_1L));
            for (int i = q_0L + 1; i <= l - q_1L; i++) {
                xiS[i] = xiS[i-1].add(einsMinustau_0Plustau1TMinusSDivQL);
            }
        } {
            /*... im Intervall [t-τ_1(t-s)],t]. */
            System.out.println();
            Dfp tau_1TMinusSDivQ_1L = tau_1.multiply(TMinusS).divide(q_1L);
            for (int i = l - q_1L + 1; i <= l; i++) {
```

```
                xiS[i] = xiS[i-1].add(tau_1TMinusSDivQ_1L);
            }
        }
    }

    /**
     * Gibt den i-ten Gitterknoten zurück, i=0,...,l.
     * @return xi[i]
     */
    public Dfp getXi(int i) {
        return xiS[0].newInstance(xiS[i]);
    }

    /**
     * Gibt eine Kopie des Feldes Dfp[] xiS der Gitterknoten zurück.
     * @return eine Kopie von Dfp[] xiS
     */
    public Dfp[] getXi() {
        return xiS.clone();
    }

    /**
     * Gibt eine Kopie von τ zurück.
     * @return eine Kopie von τ
     */
    public Dfp getTau() {
        return tau.newInstance(tau);
    }

    /**
     * Gibt eine Kopie von τ_0 zurück.
     * @return eine Kopie von τ_0
     */
    public Dfp getTau_0() {
        return tau_0.newInstance(tau_0);
    }

    /**
     * Gibt eine Kopie von τ_1 zurück.
     * @return eine Kopie von τ_1
     */
    public Dfp getTau_1() {
        return tau_1.newInstance(tau_1);
    }
}
```

A.6 BakhvalovGitter.java

```
import org.apache.commons.math3.analysis.RealFieldUnivariateFunction;
import org.apache.commons.math3.dfp.Dfp;

/**
```

```
 * Repräsentiert ein BAKHVALOV-Gitter auf einem beliebigen Intervall [s,t] mit
 * l Teilintervallen in Form eines Feldes Dfp [], welches die Knoten enthält.
 */
public class BakhvalovGitter implements GrenzschichtAngepasstesGitter<Dfp> {

    /** Enthält die Knoten. */
    private final Dfp[] xiB;
    /**
     * Enthält die Überganspunkte τ:
     */
    private Dfp tau, tau_0, tau_1;
    /**
     * Erzeugt ein BAKHVALOV-Gitter ξ^S = (ξ_i^S) mit l+1 Knoten für eine
     * Konvektionsdiffusionsgleichung.
     * @param l die Anzahl der Teilintervalle.
     * @param s der erste Gitterknoten ξ_0^S.
     * @param t der letzte Gitterknoten ξ_l^S.
     * @param q ein Gitterparamter beschreibt größenordnungsmäßig den Anteil
     * der Gitterknoten, welcher in der Grenzschicht liegt.
     * @param sigma ein Gitterparamter beschreibt die Auflösung der
     * Grenzschicht und wird typischerweise nahe der formalen
     * Konvergenzordnung der verwendeten Methode gewählt.
     * @param beta gegeben durch die Koeffzienten der Differentialgleichung
     * -εu'' - bu' + cu = f, für die das Gitter erzeugt wird, durch b ≥ β > 0.
     * @param epsilon singulärer Störungsparamter für welchen das Gitter
     * erzeugt werden soll.
     */
    public BakhvalovGitter(int l, final Dfp s, final Dfp t, final Dfp q,
            Dfp sigma, Dfp beta, Dfp epsilon) {
        xiB = new Dfp[l+1];
        xiB[0] = s;
        /* Wiederholt auftretender Ausdruck */
        final Dfp sigmaEpsilonDivBeta = sigma.multiply(epsilon).divide(beta);
        /*
         * Die gittererzeugende Funktion χ(r) := (-σε/β) ln((q-r)/q) für r = [0,τ]
         * innerhalb der Grenzschicht.
         */
        RealFieldUnivariateFunction<Dfp> chi = new
                RealFieldUnivariateFunction<Dfp>() {
            private final Dfp sigEpsDivBet = sigmaEpsilonDivBeta;
            public Dfp value(Dfp x) {
                return sigEpsDivBet.multiply(q.subtract(x).divide(q)
                        .log()).negate();
            }
        };
        /*
         * Bestimmt τ iterativ als Lösung der Gleichung χ'(τ_{i+1}) = (1-χ(τ_i))/(1-τ_i), die
         * genau dann existiert, wenn σε < βq. Andernfalls ist tau = 0 und
         * das Gitter global uniform.
         */
        tau = s.getZero();
```

```
        if (sigma.multiply(epsilon).lessThan(beta.multiply(q))) {
            Dfp tau_i;
            do {
                tau_i = tau;
                tau = q.subtract(sigmaEpsilonDivBeta.multiply(q.getOne()
                        .subtract(tau_i)).divide(q.getOne().subtract(chi
                                .value(tau_i))));
            } while (!(tau.subtract(tau_i).isZero()));
        }
        Dfp TMinusS = t.subtract(s), chiTau = chi.value(tau);
        for (int i = 1; i <= l; i++) {
            Dfp r_i = s.newInstance(i).divide(l);
            /* Bestimmung der ξ_i mittels χ, falls r_i<τ, ...*/
            if (r_i.lessThan(tau)) {
                xiB[i] = s.add(chi.value(r_i).multiply(TMinusS));
//                System.out.println("xiB[" + i + "] = " + xiB[i]);
            } else {
//                System.out.println("r_" + i + " = " + r_i);
                /*
                 * ... und die Berechnung des ersten Gitterknotens außerhalb
                 * des Intervalls [0,τ] gefolgt von der additiven Bestimmung
                 * aller folgenden Knoten des uniformen Teilgitters.
                 */
                xiB[i] = s.add(chiTau.add(sigmaEpsilonDivBeta
                        .divide(q.subtract(tau)).multiply(r_i
                        .subtract(tau))).multiply(TMinusS));
                Dfp temp = t.subtract(xiB[i]).divide(l-i);
                for (int j = i+1; j <= l; j++) {
                    xiB[j] = xiB[j-1].add(temp);
//                    System.out.println("xiB[" + j + "] = " + xiB[j]);
                }
                break;
            }
        }
    }

    /**
     * Gibt den i-ten Gitterknoten zurück, i=0,...,l.
     * @return xi[i]
     */
    public Dfp getXi(int i) {
        return xiB[0].newInstance(xiB[i]);
    }

    /**
     * Gibt eine Kopie des Feldes Dfp[] xiB der Gitterknoten
     * im Intervall [s,t] zurück.
     * @return eine Kopie von Dfp[] xiB
     */
    public Dfp[] getXi() {
        return xiB.clone();
```

```
    }

    /**
     * Gibt eine Kopie von τ zurück.
     * @return eine Kopie von τ
     */
    public Dfp getTau() {
        return tau.newInstance(tau);
    }

    /**
     * Gibt eine Kopie von τ_0 zurück.
     * @return eine Kopie von τ_0
     */
    public Dfp getTau_0() {
        return tau_0.newInstance(tau_0);
    }

    /**
     * Gibt eine Kopie von τ_1 zurück.
     * @return eine Kopie von τ_1
     */
    public Dfp getTau_1() {
        return tau_1.newInstance(tau_1);
    }
}
```

A.7 SingulaerGestoerterFall.java

```
import org.apache.commons.math3.analysis.RealFieldUnivariateFunction;
import org.apache.commons.math3.dfp.Dfp;
import org.apache.commons.math3.dfp.DfpField;

/**
 * Repräsentiert den singulär gestörten Fall und behandelt das Verfahren auf
 * den verschiedenen Gittern.
 */
public class SingulaerGestoerterFall extends BezierKollokation {

    /**
     * Erzeugt eine Instanz im singulär gestörten Fall. Dabei wird der
     * allgemeine Konstruktor unverändert aufgerufen, aber die ermittelte
     * Näherungslösung in den statischen Methoden der Klasse anders behandelt.
     * @param k Anzahl der Kollokationspunkte je Gitterintervall.
     * @param xi das Gitter auf dem die Näherungslösung berechnet werden soll.
     * @param epsilon singulärer Störungsparameter.
     * @param eta1 linker Randwert.
     * @param eta2 rechter Randwert.
     * @param p Koeffizientenfunktion in −εy'' − py' + qy = f.
     * @param q Koeffizientenfunktion in −εy'' − py' + qy = f.
     * @param f in −εy'' − py' + qy = f.
     */
```

```
public SingulaerGestoerterFall(int k, Gitter<Dfp> xi, Dfp epsilon,
        Dfp eta1, Dfp eta2, RealFieldPolynomialFunction p,
        RealFieldUnivariateFunction<Dfp> q,
        RealFieldUnivariateFunction<Dfp> f) {
    super(k, xi, epsilon, eta1, eta2, p, q, f);
}

/**
 * Erzeugt für ein übergebenes k und ein Feld l Instanzen des
 * Kollokationsverfahrens für das Reaktionsdiffusionsproblem und wertet
 * diese in Abhängigkeit von mode aus.
 * @param k die Anzahl der Kollokationspunkte je Gitterintervall.
 * @param l Feld mit den interessierenden Anzahlen von Gitterintervallen.
 * @param mode Gibt an, welche Testdaten auf der Konsole ausgegeben werden
 * sollen:
 * mode = 1: Berechnet die Fehler und die experimentelle
 * Konvergenzordnung für das Reaktionsdiffusionsproblem.
 * mode = 2: Gibt die Maplebefehle zum Plotten der Näherungslösung aus.
 * mode = 3: Gibt die Abweichungen von den Kollokations- und
 * Randbedingungen aus.
 * @return Ausgabe gemäß mode in Form eines formatierten String.
 */
public static String berechneReaktion (int k, int[] l, final Dfp epsilon,
        int mode) {
    /* Der linke Rand s des Kollokationsintervalls [s,t]. */
    final Dfp s = epsilon.getZero();
    /* Der rechte Rand t des Kollokationsintervalls [s,t]. */
    final Dfp t = epsilon.getOne();
    /* Der Randwert y(s) = η1. */
    final Dfp eta1 = epsilon.getZero();
    /* Der Randwert y(t) = η2. */
    final Dfp eta2 = epsilon.getZero();
    /* Die Koeffizientenfunktion p in -ε²y'' - py' + qy = f. */
    final RealFieldPolynomialFunction p = new RealFieldPolynomialFunction(
            new Dfp[] {epsilon.getZero()});
    /* Die Koeffizientenfunktion q in -ε²y'' - py' + qy = f. */
    final RealFieldUnivariateFunction<Dfp> q = new
            RealFieldUnivariateFunction<Dfp>() {

        public Dfp value(Dfp x) {
            return x.cos().add(x.multiply(x)).add(x.getOne());
        }
    };
    /* Die Funktion f in -ε²y'' - py' + qy = f. */
    final RealFieldUnivariateFunction<Dfp> f = new
            RealFieldUnivariateFunction<Dfp>() {
        public Dfp value(Dfp x) {
            return x.pow(9d/2).add(x.sin());
        }
    };
    /* Häufig benötigte Ausdrücke. */
```

```
Dfp qu_0 = epsilon.getTwo().pow(2).reciprocal(),
        qu_1 = qu_0,
        sigma_0 = epsilon.getTwo(), sigma_1 = sigma_0;
String ausgabe = "k = " + k + ", ";
switch (mode) {
case 1:
    /*
     * max[i] enthält die maximale Abweichung zur Vergleichslösung
     * bei l[i] Gitterintervallen.
     */
    Dfp[] max = new Dfp[l.length];
    for (int i = 0; i < l.length; i++) {
        max[i] = epsilon.getZero();
    }
    ausgabe += "epsilon = 10^" + ((int) (epsilon.log().toDouble()
            /Math.log(10)))+ ": " + "\n";
    for (int i = 0; i < l.length; i++) {
        Gitter<Dfp> originalGitter = new ShishkinGitter(l[i],s,t,qu_0,
                qu_1, sigma_0, sigma_1, epsilon.getOne(), epsilon);
        max[i] = berechneAbweichung(k, originalGitter,
                epsilon.pow(2), eta1, eta2, p, q, f);
        ausgabe += "l = ";
        for (int m = 0; m < 3 - Integer.toString(l[i]).length(); m++){
            ausgabe += " ";
        };
        ausgabe += l[i] + ": E_\\xi = " + max[i] + "\n";
        /*
         * Berechnet die experimentelle Konvergenzordnung wie in der
         * Ausarbeitung angegeben.
         */
        if (i > 0) {
                for (int m = 0; m < 2-Integer.toString(l[i]).length();
                        m++) {
                    ausgabe += " ";
                }
                ausgabe += "   \\alpha_" + l[i] + " = " +max[i].divide
                        (max[i-1]).log().divide(Math.log(Math.log(l[i]
                                ) / ( 2*Math.log(l[i-1])))) + "\n";
        }
        ausgabe += "\n";
    }
    break;
case 2:
    for (int i = 0; i < l.length; i++) {
        Gitter<Dfp> gitter = new ShishkinGitter(l[i], s, t, qu_0,
                qu_1, sigma_0, sigma_1, epsilon.getTwo(), epsilon);
        final SingulaerGestoerterFall kollokation = new
                SingulaerGestoerterFall(k, gitter, epsilon.pow(2),
                        eta1, eta2, p, q, f);
        final BezierSplineFunction g = kollokation.getG();
        String werte = "", stellen = "";
```

```
            for (int j = 0; j < l[i]; j++) {
                werte += g.value(kollokation.getXi(j)).toDouble();
                stellen += kollokation.getXi(j).toDouble();
                werte += ",";
                stellen += ",";
            }
            werte += g.value(kollokation.getXi(l[i])).toDouble();
            stellen += kollokation.getXi(l[i]).toDouble();
            ausgabe = "u" + k + "_" + l[i] + "_" + Math.abs((int) (epsilon
                    .log().toDouble() / Math.log(10))) + ":=pointplot([" +
                    stellen + "],[" + werte + "],legend=\"u" + k +
                    "_" + l[i] + "_" + Math.abs((int) (epsilon.log()
                            .toDouble() / Math.log(10))) +"\",color=\""
                    +Linienfarbe.values()[i%(Linienfarbe.values().length)]
                            + "\",connect=true):";
        }
        break;
    case 3:
        for (int i = 0; i < l.length; i++) {
            Gitter<Dfp> gitter = new ShishkinGitter(l[i], s, t, qu_0,
                    qu_1, sigma_0, sigma_1, epsilon.getTwo(), epsilon);
                    final SingulaerGestoerterFall kollokation = new
                    SingulaerGestoerterFall(k, gitter, epsilon.pow(2),
                            eta1, eta2, p, q, f);
            final BezierSplineFunction g = kollokation.getG();
            ausgabe += "g(" + s + ")=" + g.value(s) + "\n";
            for (int j = 1; j <= l[i] * k; j++) {
                Dfp temp = kollokation.getTau(j);
                ausgabe += "tau" + j + "=" + temp.toDouble() +
                        ":-epsilon*g''-p*g'+q*g-f=" +
                        (epsilon.pow(2).negate().multiply(
                                g.derivative(temp, 2)).subtract(p
                                        .value(temp)
                                        .multiply(g.derivative(temp, 1)))
                                .add(q.value(temp).multiply
                                        (g.derivative(temp, 0)))
                                .subtract(f.value(temp)))
                        + "\n";
            }
            ausgabe += "g(" + t + ")=" + g.value(t);
        }
    }
    return ausgabe;
}

/**
 * Erzeugt für die übergebenen Daten eine Vergleichslösung auf einem 7-mal
 * feineren Gitter und berechnet die maximale Abweichung der
 * Funktionswerte an den Gitterknoten des Originals.
 * @param k Anzahl der Kollokationspunkte je Gitterintervall.
 * @param xi das Gitter auf dem die Originalnäherung berechnet wurde.
```

```
     * @param epsilon singulärer Störungsparameter.
     * @param eta1 linker Randwert.
     * @param eta2 rechter Randwert.
     * @param p Koeffizientenfunktion in -εy'' - py' + qy = f.
     * @param q Koeffizientenfunktion in -εy'' - py' + qy = f.
     * @param f in -εy'' - py' + qy = f.
     */
    private static Dfp berechneAbweichung (int k, Gitter<Dfp> originalGitter,
            Dfp epsilon, Dfp eta1, Dfp eta2, RealFieldPolynomialFunction p,
            RealFieldUnivariateFunction<Dfp> q,
            RealFieldUnivariateFunction<Dfp> f) {
        Dfp max = epsilon.getZero();
        Gitter<Dfp> modifiziertesGitter = new ModifiziertesGitter(
                originalGitter,7);
        final SingulaerGestoerterFall originalKollokation = new
                SingulaerGestoerterFall(k, originalGitter, epsilon,
                        eta1, eta2, p, q, f), modifizierteKollokation = new
                        SingulaerGestoerterFall(k,modifiziertesGitter,
                                epsilon, eta1, eta2, p, q, f);
        final BezierSplineFunction originalG = originalKollokation
                .getG(),modifiziertesG=modifizierteKollokation.getG();
        for (int j = 0; j <= originalGitter.getXi().length-1; j++) {
            Dfp temp = originalG.value(originalGitter.getXi(j))
                    .subtract(modifiziertesG.value(originalGitter
                            .getXi(j))).abs();
            if (temp.greaterThan(max)) {
                max = temp;
            }
        }
        return max;
    }

    /**
     * Erzeugt für ein übergebenes k und ein Feld l Instanzen des
     * Kollokationsverfahrens für das Konvektionsdiffusionsproblem und wertet
     * diese in Abhängigkeit von mode aus.
     * @param k die Anzahl der Kollokationspunkte je Gitterintervall.
     * @param l Feld mit den interessierenden Anzahlen von Gitterintervallen.
     * @param mode Gibt an, welche Testdaten auf der Konsole ausgegeben werden
     * sollen:
     * mode = 11: Berechnet die Fehler und die experimentelle
     * Konvergenzordnung für das Konvektionsdiffusionsproblem auf einem
     * Shishkin-Gitter.
     * mode = 12: Berechnet die Fehler und die experimentelle
     * Konvergenzordnung für das Konvektionsdiffusionsproblem auf einem
     * Bakhvalov-Gitter.
     * mode = 2: Gibt die Maplebefehle zum Plotten der Näherungslösung aus.
     * mode = 3: Gibt die Abweichungen von den Kollokations- und
     * Randbedingungen aus.
     * @return Ausgabe gemäß mode in Form eines formatierten String.
     */
```

```
public static String berechneKonvektion (int k, int[] l, final
        Dfp epsilon, int mode) {
    /* Der linke Rand s des Kollokationsintervalls [s,t]. */
    final Dfp s = epsilon.getZero();
    /* Der rechte Rand t des Kollokationsintervalls [s,t]. */
    final Dfp t = epsilon.getOne();
    /* Der Randwert y(s) = η1. */
    final Dfp eta1 = epsilon.getZero();
    /* Der Randwert y(t) = η2. */
    final Dfp eta2 = epsilon.getZero();
    /* Die Koeffizientenfunktion p in -εy'' - py' + qy = f. */
    final RealFieldPolynomialFunction p = new RealFieldPolynomialFunction(
            new Dfp[] {epsilon.getOne()});
    /* Die Koeffizientenfunktion q in -εy'' - py' + qy = f. */
    final RealFieldPolynomialFunction q = new RealFieldPolynomialFunction(
            new Dfp[] {epsilon.getTwo()});
    /* Die Funktion f in -εy'' - py' + qy = f. */
    final RealFieldUnivariateFunction<Dfp> f = new
            RealFieldUnivariateFunction<Dfp>() {
        public Dfp value(Dfp x) {
            return x.subtract(epsilon.getOne()).exp();
        }
    };
    /* Häufig benötigter Ausdruck. */
    Dfp qu = epsilon.getTwo().reciprocal(), sigma = epsilon.getTwo();
    Dfp[] max;
    String ausgabe = "k = " + k + ", ";
    switch (mode) {
    case 11:
        /*
         * max[i] enthält die maximale Abweichung zur Vergleichslösung
         * bei l[i] Gitterintervallen.
         */
        max = new Dfp[l.length];
        for (int i = 0; i < l.length; i++) {
            max[i] = epsilon.getZero();
        }
        ausgabe += "epsilon = 10^" + ((int) (Math.log(epsilon.toDouble())
                /Math.log(10)))+ ": " + "\n";
        for (int i = 0; i < l.length; i++) {
            Gitter<Dfp> originalGitter = new ShishkinGitter(l[i], s, t,
                    qu, sigma, epsilon.getTwo().reciprocal().pow(2), epsilon);
            max[i] = berechneAbweichung(k, originalGitter, epsilon,
                    eta1, eta2, p, q, f);
            ausgabe += "l = ";
            for (int m = 0; m < 3 - Integer.toString(l[i]).length(); m++){
                ausgabe += " ";
            };
            ausgabe += l[i] + ": E_\\xi = " + max[i] + "\n";
            if (i > 0) {
                    for (int m = 0; m < 2-Integer.toString(l[i]).length();
```

```
                        m++) {
                    ausgabe += " ";
                };
                ausgabe += "    \\alpha_" + l[i] + " = " +
                        max[i].divide(max[i-1]).log().divide(
                                Math.log(Math.log(l[i])/(2*
                                        Math.log(l[i-1])))) +"\n";
            }
            ausgabe += "\n";
        }
        break;
    case 12:
        /*
         * max[i] enthält die maximale Abweichung zur Vergleichslösung
         * bei l[i] Gitterintervallen.
         */
        max = new Dfp[l.length];
        for (int i = 0; i < l.length; i++) {
            max[i] = epsilon.getZero();
        }
        ausgabe += "epsilon = 10^" + ((int) (Math.log(epsilon.toDouble())
                /Math.log(10)))+ ": " + "\n";
        for (int i = 0; i < l.length; i++) {
            Gitter<Dfp> originalGitter = new BakhvalovGitter(l[i], s, t,
                    qu, sigma, epsilon.getTwo().reciprocal().pow(2),
                    epsilon);
            max[i] = berechneAbweichung(k, originalGitter, epsilon,
                    eta1, eta2, p, q, f);
            ausgabe += "l = ";
            for (int m = 0; m < 3 - Integer.toString(l[i]).length(); m++){
                ausgabe += " ";
            };
            ausgabe += l[i] + ": E_\\xi = " + max[i] + "\n";
            if (i > 0) {
                    for (int m = 0; m < 2-Integer.toString(l[i]).length();
                            m++) {
                        ausgabe += " ";
                    };
                    ausgabe += "    \\alpha_" + l[i] + " = " +
                            max[i-1].divide(max[i]).log().divide(
                                    Math.log(2)) + "\n";
            }
            ausgabe += "\n";
        }
        break;
    case 2:
        for (int i = 0; i < l.length; i++) {
            Gitter<Dfp> gitter = new BakhvalovGitter(l[i], s, t, qu,
                    sigma, epsilon, epsilon);
            final SingulaerGestoerterFall kollokation = new
                    SingulaerGestoerterFall(k, gitter, epsilon, eta1,
```

```
                        eta2, p, q, f);
            final BezierSplineFunction g = kollokation.getG();
            String werte = "", stellen = "";
            for (int j = 0; j < l[i]; j++) {
                werte += g.value(kollokation.getXi(j)).toDouble();
                stellen += kollokation.getXi(j).toDouble();
                werte += ",";
                stellen += ",";
            }
            werte += g.value(kollokation.getXi(l[i])).toDouble();
            stellen += kollokation.getXi(l[i]).toDouble();
            ausgabe = "g" + k + "_" + l[i] + "_" + Math.abs((int) (epsilon
                    .log().toDouble() / Math.log(2))) + ":=pointplot([" +
                    stellen + "],[" + werte + "],legend=\"u" + k +
                    "_" + l[i] + "_" + Math.abs((int) (epsilon.log()
                            .toDouble() / Math.log(10)))+"\",color=\""+
                            Linienfarbe.values()[i]+"\",connect=true):";
        }
        break;
    case 3:
        for (int i = 0; i < l.length; i++) {
            Gitter<Dfp> gitter = new BakhvalovGitter(l[i], s, t, qu,sigma,
                    epsilon, epsilon);
                    final SingulaerGestoerterFall kollokation = new
                    SingulaerGestoerterFall(k, gitter, epsilon, eta1,
                            eta2, p, q, f);
            final BezierSplineFunction g = kollokation.getG();
            ausgabe += "g(" + s + ")=" + g.value(s) + "\n";
            for (int j = 1; j <= l[i] * k; j++) {
                Dfp temp = kollokation.getTau(j);
                ausgabe += "tau" + j + "=" + temp.toDouble() +
                        ":-epsilon*g''-p*g'+q*g-f=" +
                        (epsilon.negate().multiply(g.derivative(temp, 2))
                                .subtract(p.value(temp)
                                        .multiply(g.derivative(temp, 1)))
                                .add(q.value(temp).multiply
                                        (g.derivative(temp, 0)))
                                .subtract(f.value(temp)))
                        + "\n";
            }
            ausgabe += "g(" + t + ")=" + g.value(t);
        }
    }
    return ausgabe;
}

/**
 * Berechnen verschiedener Daten in Abhängigkeit des angegebenen Modus
 * und Ausgabe auf der Konsole.
 */
public static void main (String[] args) {
```

```
        DfpField koerper = new DfpField(BezierKollokation.getGenauigkeit());
        int kleinstesL = 7, groesstesL = 9, deltaL = groesstesL - kleinstesL;
        int[] l = new int[deltaL + 1];
        l[0] = (int) Math.pow(2, kleinstesL);
        for (int i = 1; i <= deltaL; i++) {
            l[i] = l[i-1] * 2;
        }
        Dfp[] epsilon = new Dfp[] {koerper.newDfp(10).pow(-4),
                koerper.newDfp(10).pow(-8), koerper.newDfp(10).pow(-12)};
        for (int i = 0; i < epsilon.length; i++) {
            for (int k : new int[] {1}) {
              System.out.println(berechneReaktion(k, l, epsilon[i], 1));
//                System.out.println(berechneKonvektion(k, l, epsilon[i], 12));
            }
        }
    }
}
```

A.8 Binomialkoeffizient.java

```
/** Repräsentiert für einen Paramter n alle Werte von n über k,k=0,...,n.*/
public class Binomialkoeffizient {

    /** Enthält den Wert n. */
    private final int n;
    /** Enthält die berechneten Werte. */
    private final int[] binom;

    /**
     * Erzeugt eine Instanz, die alle entsprechenden Werte
     * n über k,k=0,...,n vorhält.
     */
    public Binomialkoeffizient(int n) {
        this.n = n;
        binom = new int[n+1];
        /**
         * Bei der Berechnung der Werte wird die Symmetrie des
         * Binomialkoeffizienten ausgenutzt und für k>n-k der
         * bereits berechnete Wert für n-k genutzt.
         */
        for (int k = 0; k < binom.length; k++) {
            if (k > n - k)
                binom[k] = binom[n - k];
            else
                binom[k] = berechneNUeberK(k);
        }
    }

    /**
     * Berechnet die Werte des Binomialkoeffizienten mit der Variablen
     * n und dem übergebenen Paramter.
     * @param k für das n über k berechnet werden soll.
```

```
     * @return n über k
     */
    private int berechneNUeberK(int k) {
        int b = 1;
        for (int j = 1, m = n; j <= k; j++, m--)
            b = b * m/j;
        return b;
    }

    /**
     * Gibt den Wert n über k zurück.
     * @param k für das n über k zurückgegeben werden soll.
     * @return n über k.
     */
    public int getUeber(int k) {
        return binom[k];
    }

    /**
     * Gibt alle Werte n über k,k=0,...,n zurück.
     * @param k für das n über k zurückgegeben werden soll.
     * @return double[] (\binom{n}{0},...,\binom{n}{n}).
     */
    public int[] getBinom() {
        return binom.clone();
    }

    /**
     * Gibt den Wert zurück, für den die Werte des Binomialkoeffizienten
     * abgelegt sind.
     * @return n, für das alle n über k,k=0,...,n abgelegt
     * sind.
     */
    public int getN() {
        return n;
    }
}
```

Literatur

[1] Ahlberg, J. H. und T. Ito: *A Collocation Method for Two-Point Boundary Value Problems.* Mathematics of Computation, 29:761–776, 1975.

[2] Böhm, H. Prautzsch W. und M. Paluszny: *Bézier and B-Spline Techniques.* Springer, 2002.

[3] Böhm, W., G. Farin und J. Kahmann: *A survey of curve and surface methods in CAGD.* Computer Aided Geometric Design, 1:1–60, 1984.

[4] Boor, C. de: *A Practical Guide to Splines.* Springer-Verlag, 1978.

[5] Boor, C. de und B. Schwartz: *Collocation at Gaussian Points.* SIAM Journal on Numerical Analysis, 10(4):582–606, 1973.

[6] Doha, E. H., A. H. Bhrawy und M. A. Saker: *On the Derivatives of Bernstein Polynomials: An Application for the Solution of High Even-Order Differential Equations*. Boundary Value Problems, 2011(2):1–16, 2011.

[7] Farouki, R. T. und T. N. T. Goodman: *On the Optimal Stability of the Bernstein Basis*. Mathematics of Computation, 65:1553–1566, 1996.

[8] Feilmeier, M.: *Hermitesche Kollokation bei Integralgleichungen*. Computing, 15:137–146, 1975.

[9] Fröhner, A. K.: *Defektkorrekturverfahren für singulär gestörte Randwertaufgaben*. Doktorarbeit, Fakultät Mathematik und Naturwissenschaften der Technischen Universität Dresden, 2011.

[10] Hoschek, J. und D. Lasser: *Grundlagen der geometrischen Datenverarbeitung*. B. G. Teubner Stuttgart, 1992.

[11] Linss, T.: *Layer-adapted meshes for reaction-convection-diffusion problems*. Springer, 2010.

[12] Linss, T.: *Numerische Mathematik I*. FernUniversität in Hagen, Kurs 01270, Version Sommersemester 2016.

[13] Martin, R. S. und J. H. Wilkinson: *Solution of Symmetric and Unsymmetric Band Equations and the Calculation of Eigenvectors of Band Matrices*. Numerische Mathematik, 9:279–301, 1967.

[14] Martin, R.S. und J.H. Wilkinson: *The implicit QL algorithm*. Numer. Math., 12:377–383, 1968.

[15] Reinhardt, H.-J.: *Numerik gewöhnlicher Differentialgleichungen*. De Gruyter, 2012.

[16] Schumaker, L. L.: *Spline Functions Basic Theory*. John Wiley and Sons, New York, 1981.

[17] Strasser, W.: *Einführung in Computergrafik*. FernUniversität in Hagen, Kurs 01277, Version Wintersemester 2014-2015.

[18] Unger, L.: *Lineare Algebra*. FernUniversität in Hagen, Kurs 01143, Version Wintersemester 2016-2017.