Spline-Interpolation

Eric Kunze & Johanna Preuße

30. November 2018

Zusammenfassung

In dieser Aufgabe wird die Runge-Funktion sowie eine weitere Funktion im Vergleich dazu interpoliert. Als Interpolierende wird jeweils ein linearer Spline sowie eine kubischer $\mathcal{C}^1$-Spline verwendet. Nach Herleitung der Splines werden diese grafisch dargestellt und anschließend Interpolationsfehler in Abhängigkeit der Feinheit der Zerlegung des zugrunde liegenden Intervalls ausgewertet. Schließlich wird auch die Konvergenzgeschwindigkeit der Näherung gegen die zu interpolierenden Funktionen untersucht. Die dabei auftretenden Unterschiede werden am Ende diskutiert und Möglichkeiten zur Effizienzsteigerung und Verbesserung des Programms aufgezeigt.

Inhaltsverzeichnis

1 Interpolation der Runge-Funktion

Für die Runge-Funktion

$$f(x) = \frac{1}{1 + 25x^2}$$

gilt

$$f'(x) = -\frac{50x}{625x^4 + 50x^2 + 1}$$

1.1 Herleitung des linearen Splines

Sei Δ_N eine Zerlegung des Intervall $[-1, 1]$ in äquidistante Stützstellen x_k mit $k = 0, \ldots, N$ und Stützstellenabstand $h_N := 2/N$, d.h. für die Stützstellen gilt $x_k = -1 + k \cdot h_N$. Für den Polynomspline $s^1 \in \mathcal{S}_1^0(\Delta_N)$ soll $s^1|_{[x_k, x_{k+1}]} = s_k^1$ sein. Dann ist $s_k^1 \colon [x_k, x_{k+1}] \to \mathbb{R}$ für alle $k = 0, \ldots, N-1$ eine affin lineare Funktion in jedem Intervall, d.h.

$$s_k^1(x) = f(x_k) + \frac{f(x_{k+1}) - f(x_k)}{h_N} \cdot (x - x_k) \quad \text{für alle } k = 0, \ldots N$$

Die Implementierung in Octave sieht dann wie folgt aus:

```
1  function y = splinef1(N)
2    % linearer spline der runge funktion
3    % N ... feinheit, mit der Spline berechnet wird
4
5    h=2/N;
6    x=-1:h:1;
7
8    M = 10*N;
9    hM = 2/M;
10   x0=-1:hM:1;
11
12   j = 1;
13   for i = 1:N
14     for k=1:10
15       y(j) = ( ( f(x(i+1)) - f(x(i)) ) / h ) * (x0(j) -
             x(i)) + f(x(i));
16       j = j + 1;
17     end
18   end
19   y(j) = f(x(N+1));
20 end
```

Listing 1: Lineare Splineinterpolation

Dabei werden jeweils zehn Elemente der Verfeinerung M in einem Splineabschnitt ausgewertet, da $M = 10 \cdot N$ gilt. Dies beschleunigt die Berechnung insbesondere bei großen N bzw. M, da die mehrfache Berechnung gleicher Werte ausgeschlossen wird. Eine Implementierung, die für zwei unabhängige Feinheiten Ergebnisse liefert, befindet sich im entsprechenden File als Kommentar. Diese arbeitet allerdings deutlich ineffizienter.

1.2 Herleitung des kubischen Splines

Es gelten die gleichen Voraussetzungen wie in Abschnitt 1.1. Für den kubischen Spline gestaltet sich die Herleitung ein wenig schwieriger. Dazu nutzen wir die Vorgehensweise in der Vorlesung (vgl. Abschnitt 1.3 im Skript). Der Spline $s^3 \in \mathcal{S}_3^1(\Delta_N)$ setzt sich dann aus den Teilfunktionen s_k^3 zusammen, d.h. $s_3 \mid_{[x_k, x_{k+1}]} = s_k^3$. Für alle $k = 0, \dots N$ kann dann s_k^3 dargestellt werden als

$$s_k^3(x) = a_k(x - x_k)^3 + b_k(x - x_k)^2 + c_k(x - x_k) + d_k \tag{1}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} s_k^3(x) = 3a_k(x - x_k)^2 + 2b_k(x - x_k) + c_k \tag{2}$$

Aufgrund der Interpolationsbedingung gilt in Gleichung (1) $s_k(x_k) = d_k = f(x_k)$ und da s zusätzlich auch f' in allen x_k interpolieren soll mit Gleichung (2) auch $\frac{\mathrm{d}}{\mathrm{d}x} s_k^3(x_k) = c_k = f'(x_k)$. Mit der Setzung $m_k := s'(x_k) = f'(x_k)$ und $h_k = h_N$ für alle $k = 0, \dots N$ wie in Abschnitt 1.1 ergibt sich dann das lineare Gleichungssystem

$$\begin{pmatrix} h_N^3 & h_N^2 \\ 3 \cdot h_N^2 & 2 \cdot h_N \end{pmatrix} \cdot \begin{pmatrix} a_k \\ b_k \end{pmatrix} = \begin{pmatrix} f(x_{k+1}) - f(x_k) - m_k \cdot h_N \\ m_k \cdot h_N - m_k \end{pmatrix} \tag{3}$$

Die Lösung errechnet Octave selbstständig und dann erfolgt die Berechnung von $s^3(x_0)$ für ein eingegebenes x_0 wie in Abschnitt 1.1. Die Implementation sieht dann wie folgt aus:

```
1  function y = splinef3(N)
2    % kubischer c1-spline der funktion f
3    % N ... feinheit, mit der Spline berechnet wird
4
5    h=2/N;
6    x=-1:h:1;
7
8    M = 10*N;
9    hM = 2/M;
10   x0=-1:hM:1;
11
12   j = 1;
13   for i = 1:N
14       d=f(x(i));
15       c=f2(x(i));
16       ab=[h*h*h h*h;3*h*h 2*h]\[f(x(i+1))-f(x(i))-f2(x(
             i))*h;f2(x(i+1))-f2(x(i))];
17       a=ab(1);
18       b=ab(2);
19     for k=1:10
20       y(j)=d+c*(x0(j)-x(i))+b*(x0(j)-x(i))^2+a*(x0(j)-x
             (i))^3;
21       j = j + 1;
22     end
23   end
24   y(j) = f(x(N+1));
25 end
```

Listing 2: Kubische Splineinterpolation

Wiederum werden jeweils zehn Elemente der Verfeinerung M in einem Splineabschnitt ausgewertet um die Effizienz zu erhöhen. Auch hier findet sich eine allgemeinere, aber langsamere Implementierung im entsprechenden File.

1.3 Plot der Splines

Um den Spline zu plotten, wird dabei der Vektor der Verfeinerung Δ_M benutzt (für genaue Darstellung siehe Aufgabenblatt). Interessanterweise ist für den Plot des linearen Splines eigentlich gar keine explizite Berechnung des Splines notwendig, da Octave bei Übergabe der Stützstellen der Zerlegung Δ_N automatisch linear zwischen diesen interpoliert. Um die Implementierung zu testen verwenden wir jedoch die explizite Berechnung beider Splines. Der fertige Plot ist dann Abb. 1 zu entnehmen. Der Quellcode für die Plots sieht dann wie folgt aus:

```
1  % plot fuer rungefunktion und interpolierende
2  fplot = figure('Name', 'Runge-Funktion');
3  plot(x, f(x), x, splinef1(N), x, splinef3(N))
4  legend('Rungefunktion','linearer Spline','
       kubischer Spline')
5  achse = axis();
6  textpos_x = achse(1) + 0.1;
7  textpos_y = achse(4) - 0.1;
8  text(textpos_x,textpos_y, ['N=' num2str(N)])
9  print(fplot,'-dpng','fplot.png');
```

Listing 3: Plots der Runge-Funktion im Hauptprogramm

Dabei ist festzustellen, dass unsere Implementierung nicht ganz sauber ist, da an zwei verschiedenen Stellen die Zerlegungen berechnet werden: einmal als `x` im Hauptprogramm und dann jeweils als `x0` in den Spline-Funktionen. Dies ist insofern problematisch, dass Änderungen stets an beiden Stellen durchgeführt werden müssen. Jedoch soll dies für unsere Zwecke in der Form ausreichen. Soll jedoch auf der Aufgabe aufbauend weiter mit den Files gearbeitet werden, ist eine Übergabe der jeweiligen Feinheit als Parameter an die entsprechenden Spline-Funktionen einzusetzen.

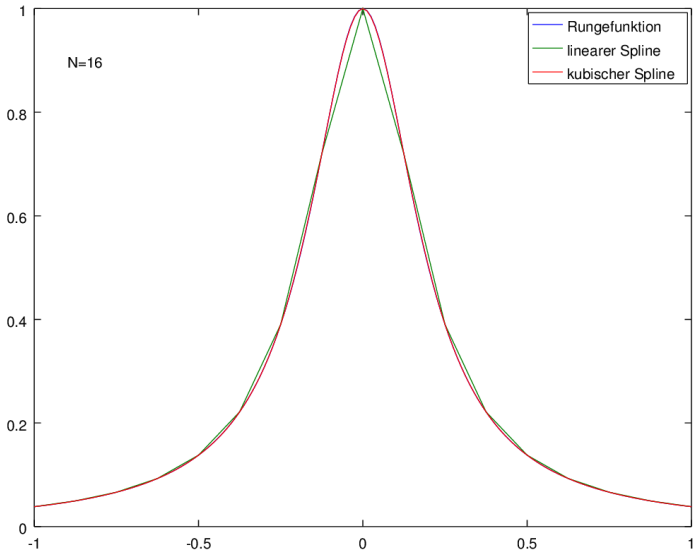

Abbildung 1: Interpolation der Runge-Funktion f

1.4 Interpolationsfehler der Runge-Funktion

Die grafische Darstellung des Interpolationsfehler stellt lediglich den Fehler an den Stellen der Zerlegung mit Feinheit M dar, vergleiche dazu Abb. 2. Die Implementierung sieht wie folgt aus.

```
1  % fehlerplot fuer rungefunktion
2  ffehlerplot = figure('Name', '
       Interpolationsfehler der Runge-Funktion');
3  plot(x,fehler(f(x),splinef1(N)), x, fehler(f(x),
       splinef3(N)))
4  legend('Fehler des linearen Splines', 'Fehler des
       kubischen Splines')
5  achse = axis();
6  textpos_x = achse(1) + 0.1;
7  textpos_y = achse(4) - 0.005;
8  text(textpos_x,textpos_y, ['N=' num2str(N)])
9  print(ffehlerplot, '-dpng','ffehlerplot.png');
```

Listing 4: Plot der Interpolationsfehler der Runge-Funktion im Hauptprogramm

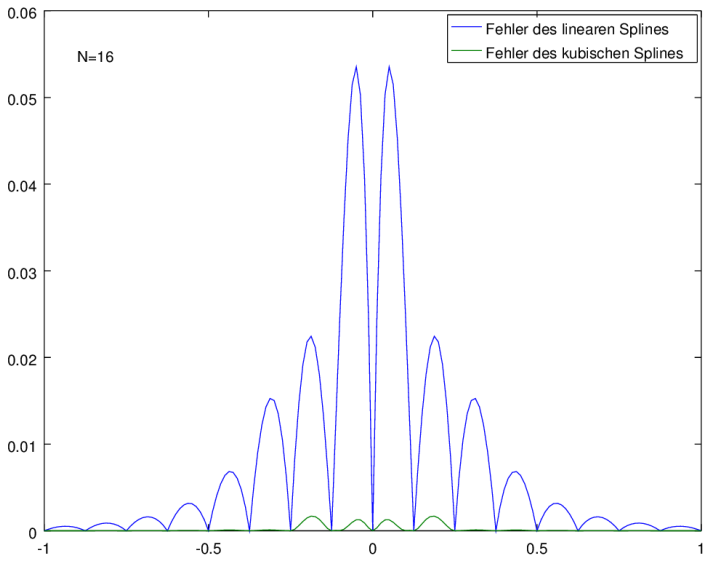

Abbildung 2: Interpolationsfehler der Runge-Funktion f

Die Berechnung der maximalen Fehler stellt den Hauptaufwand bezüglich der Rechenzeit des Programms dar. Dabei ergeben sich die Werte aus Tabelle 1.

k	N_k	$E(h_{N_k})$ linear	$E(h_{N_k})$ kubisch
0	4	0.17872	0.21938
1	8	0.063128	0.035509
2	16	0.053536	0.0016935
3	32	0.020652	0.00038860
4	64	0.0058496	0.000033560

Tabelle 1: Maximaler Fehler bei Interpolation der Runge-Funktion

Dabei erkennt man die fallende Monotonie des Fehler bezüglich der Feinheit N_k, was sich auch daraus erschließt, dass mit höherer Feinheit die Funktion besser interpoliert wird. Außerdem ist der Fehler insbesondere für höhere k beim kubischen Spline kleiner als jener beim linearen Spline. Auch dies ist klar, da kubische Splines das Krümmungsverhalten der Funktion besser annähern können (kubische Funktionen besitzen Krümmungen während lineare Funktionen stets eine Gerade beschreiben).

Für die experimentelle Konvergenzordnung ergeben sich die Werte in Tabelle 2.

k	N_k	$EOC(h_{N_k}, h_{N_{k+1}})$ linear	$EOC(h_{N_k}, h_{N_{k+1}})$ kubisch
0	4	1.5013	2.6272
1	8	0.2378	4.3901
2	16	1.3742	2.1237
3	32	1.8199	3.5334
4	64	1.9541	3.8869
5	128	1.9885	3.9719
6	256	1.9971	3.9930
7	512	1.9992	3.9982
8	1024	1.9998	3.9996
9	2048	2.0000	3.9999
10	4096	2.0000	4.0000

Tabelle 2: Experimentelle Konvergenzordnung bei Interpolation der Runge-Funktion

Die experimentelle Konvergenzordnung beschreibt die Geschwindigkeit, mit der die Interpolierende gegen die Interpolierte (hier also die Runge-Funktion) konvergiert. Dementsprechend kann man den Daten entnehmen, dass der kubische Spline für große k doppelt so schnell gegen die Runge-Funktion konvergiert wie der lineare Spline.

2 Interpolation der anderen Funktion

Für die Funktion

$$g(x) = \left(1 + \cos\left(\frac{3}{2}\pi \cdot x\right)\right)^{\frac{2}{3}}$$

gilt

$$g'(x) = -\frac{\pi \sin\left(\frac{3}{2}\pi x\right)}{\sqrt[3]{1 + \cos\left(\frac{3}{2}\pi x\right)}}$$

2.1 Interpolation durch linearen und kubischen Spline

Die Herleitung der Splines gestaltet sich exakt genauso wie für die Splines der Runge-Funktion, siehe also Abschnitt 1.1 und Abschnitt 1.2. Die Implementierung in Octave ist ebenso nahezu identisch und kann dem entsprechenden File entnommen werden. Ebenso ist auch die programmiertechnische Umsetzung der Plots (vgl. Abb. 3) analog. Der Quellcode dafür findet sich im Hauptskript.

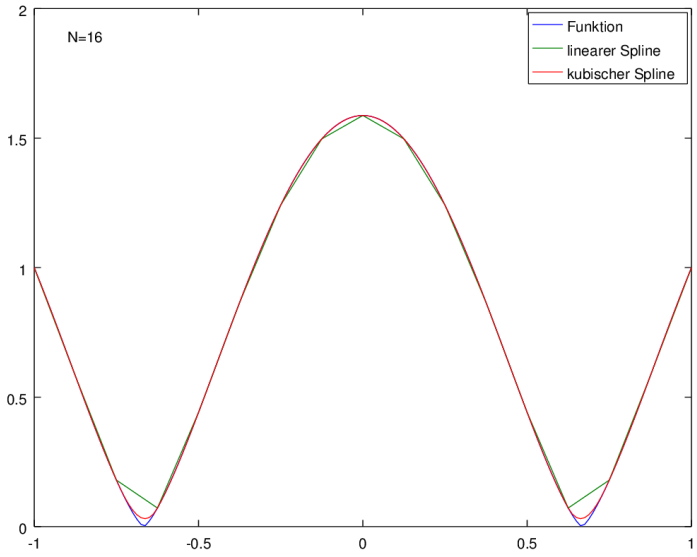

Abbildung 3: Interpolation der anderen Funktion g

2.2 Interpolationsfehler der anderen Funktion

Die grafische Darstellung des Interpolationsfehler stellt lediglich den Fehler an den Stellen der Zerlegung mit Feinheit M dar, vergleiche dazu Abb. 4. Die Implementierung ist wiederum analog zu jener in Abschnitt 1.4 und kann im entsprechenden File genauer nachgeschlagen werden.

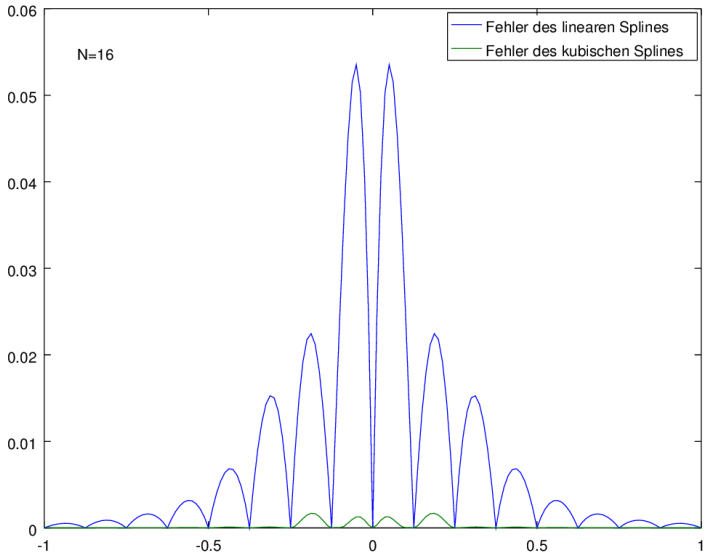

Abbildung 4: Interpolationsfehler der anderen Funktion g

Bereits in der grafischen Darstellung ist erkennbar, dass diese Funktion wahrscheinlich schlechter interpoliert werden kann, da beispielsweise in den Minima auch der kubische Spline noch erkennbar vom originalen Verlauf der Funktion g abweicht. Dies ist mit der starken Anstiegsänderung im Verhältnis zu einem relativ kleinen Intervall der Stützstellen zu erklären, sodass die starke Krümmung nicht von einer Funktion dritten Grades genähert werden kann. Dies könnte man eventuell beheben, indem man größere Polynomgrade zulässt, jedoch ergeben sich dann unter Umständen die bei Polynominterpolationen mit höhergradigen Polynomen auftretenden Oszillationen. Außerdem können wir bei linearer und kubischer Splineinterpolation eine für große k gleiche experimentelle Konvergenzordnung feststellen (vgl Tabelle 4). Damit bringt also auch der kubische Spline keinen Vorteil in der Geschwindigkeit, mit der man sich der Originalfunktion nähert. Mit einem Wert von $EOC \approx 1.\overline{3}$ ist diese außerdem geringer als bei der Runge-Funktion.

k	N_k	$E(h_{N_k})$ linear	$E(h_{N_k})$ kubisch
0	4	0.61130	0.19577
1	8	0.26300	0.070736
2	16	0.10648	0.027316
3	32	0.042468	0.010764
4	64	0.016874	0.0042640

Tabelle 3: Maximaler Fehler bei Interpolation der anderen Funktion

k	N_k	$EOC(h_{N_k}, h_{N_{k+1}})$ linear	$EOC(h_{N_k}, h_{N_{k+1}})$ kubisch
0	4	1.2168	1.4686
1	8	1.3045	1.3727
2	16	1.3261	1.3436
3	32	1.3316	1.3359
4	64	1.3328	1.3340
5	128	1.3332	1.3335
6	256	1.3332	1.3334
7	512	1.3333	1.3333
8	1024	1.3333	1.3333
9	2048	1.3333	1.3333
10	4096	1.3333	1.3333

Tabelle 4: Experimentelle Konvergenzordnung bei Interpolation der anderen Funktion

3 Auswertung der Ergebnisse

Nachdem beide Funktionen jeweils linear und kubisch interpoliert worden sind ergeben sich die folgenden Ergebnisse.

Offensichtlich eignet sich die Runge-Funktion f besser zur Spline-Interpolation als die Funktion g. Dies erkennt man insbesondere an den kleineren maximalen Fehlern und der höheren Konvergenzgeschwindigkeit. Das heißt also im Allgemeinen braucht man weniger Ressourcen (weniger Zeit/Rechenaufwand aufgrund schnellerer Konvergenz) und erhält damit bessere Interpolationsergebnisse. Dies kann auf die unterschiedlichen Krümmungseigenschaften der beiden Funktionen zurückgeführt werden, denn die Runge-Funktion besitzt lediglich ein Maximum, wohingegen die andere Funktion g noch zwei weitere Minima besitzt, in deren Umgebung sich der Tangentenanstieg relativ schnell ändert.

Hinsichtlich der Implementierung können wir resümieren, dass in unserem Quellcode sicher noch einige Optimierungen getätigt werden können, insbesondere in Hinblick auf die Laufzeit. Diese ist doch recht lang, inbesondere die Berechnung der maximalen Fehler ist sehr rechenaufwändig. Des Weiteren sollten für die bessere Nachnutzbarkeit auch die in Abschnitt 1.3 angesprochenen Veränderungen durchgeführt werden. Allerdings liefert unser Programm bereits zuverlässig genau Werte, mit denen man in diesem Rahmen durchaus arbeiten kann.