Interpolation der Runge-Funktion und anderer Funktionen mit Octave

Henry Haustein, Lars Ortscheidt

28. November 2018

Inhaltsverzeichnis

1 Interpolation der Runge-Funktion

$$f(x) = \frac{1}{1+25x^2}$$
$$f'(x) = -\frac{50x}{625x^4+50x^2+1}$$

1.1 Berechnung der Splines

1.1.1 Polynomsplines aus $\mathcal{S}_1^0(\Delta)$

Eine Polynomspline $s \in \mathcal{S}_1^0(\Delta)$ ist eine affin lineare Funktion, das heißt er hat die Form $s(x) = mx + n$ mit Anstieg m und y-Achsenverschiebung n.

Die Interpolationsfunktion g_N, mit $N + 1$ Stützstellen, besteht nun also aus Splines $s_i \in \mathcal{S}_1^0(\Delta)$, wobei für jeden Spline gilt:

$$\text{Definitionsbereich: } [x_i, x_{i+1}]$$
$$m_i = \frac{f_{i+1} - f_i}{x_{i+1} - x_i}$$
$$n_i = f_i$$

wobei x_i die Stützstellen und f_i die Stützwerte sind. Dabei läuft i von 0 bis $N - 1$.

Der Quelltext für Octave sieht dann so aus:

```
runge = @(x) 1./(1+25*x.^2);
xreal = -1:0.01:1;

n = input('Anzahhl der Stuetzstellen - 1 := N: ');

%Schritweite h berechnen
h = 2/n
%Stuetzstellenvektor x berechnen
x = -1:h:1;

for i=1:n+1
 %Stutzwertevektor f berechnen
 f(i) = runge(x(i));
endfor

for i=1:n
 %Anstiege m_i berechnen
 m(i) = (f(i+1)-f(i))./(x(i+1)-x(i));
 %Achsenabschnitte n_i berechnen
```

```
 n(i) = f(i);
endfor

plot(x, f, "-;Interpol.;", xreal, runge(xreal), "-;Rungefkt.;")
```

Das Interessante hierbei ist, dass die berechneten Werte in den Arrays `m` und `n` gar nicht für die Interpolation gebraucht werden - die Funktion `plot` interpoliert automatisch linear, wenn man ihr die Stützstellen und -werte übergibt.

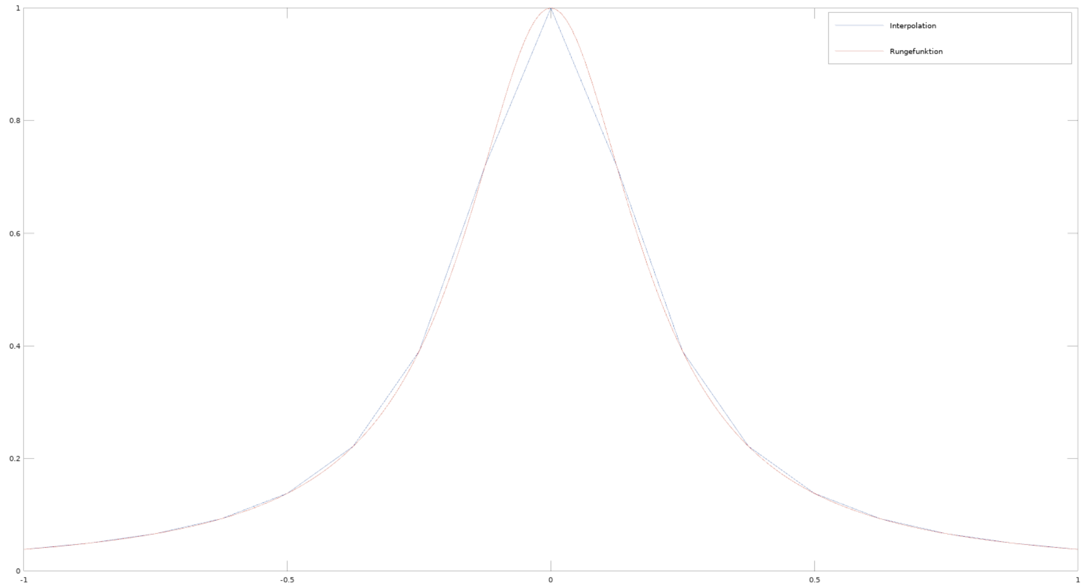

Abbildung 1: lineare Splineinterpolation mit $N = 16$

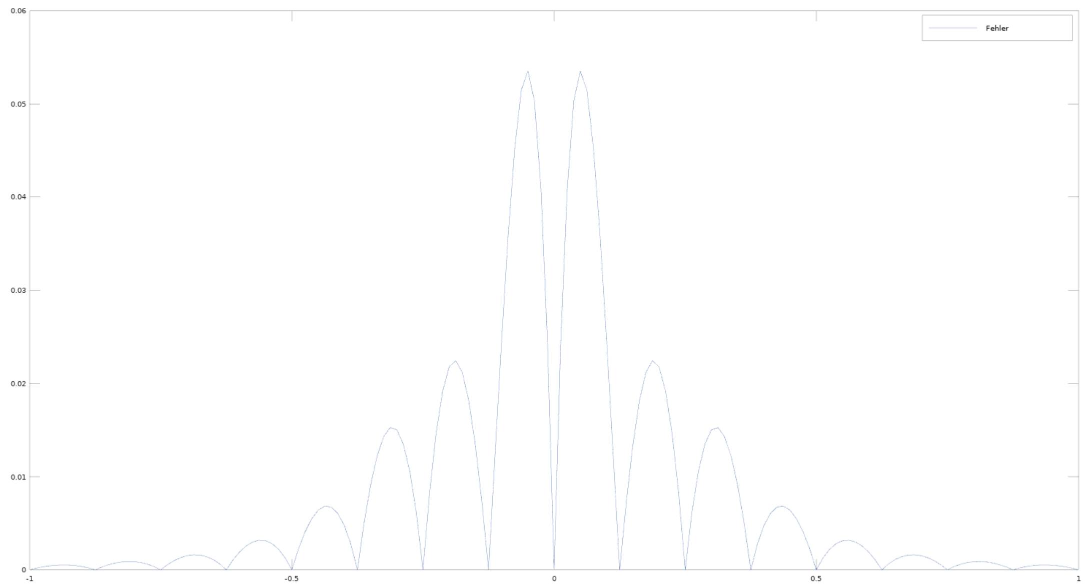

Abbildung 2: Fehler bei linearer Splineinterpolation mit $N = 16$

1.1.2 Polynomsplines aus $\mathcal{S}_3^1(\Delta)$

Das die Interpolationssplines Polynome dritten Grades und einmal stetig differenzierbar sein sollen, nehmen wird aus der Vorlesung den Ansatz (1.7):

$$s_k(x) = a_k(x - x_k)^3 + b_k(x - x_k)^2 + c_k(x - x_k) + d_k$$

mit $x \in [x_k, x_{k+1}]$. Die Vorfaktoren a_k, b_k, c_k und d_k ergeben sich aus (1.9) und (1.10) in der Vorlesung.

$$\begin{aligned} d_k &= f_k \\ c_k &= m_k = s'(x_k) = f'(x_k) \end{aligned}$$

$$\begin{pmatrix} h_k^3 & h_k^2 \\ 3h_k^2 & 2h_k \end{pmatrix} \begin{pmatrix} a_k \\ b_k \end{pmatrix} = \begin{pmatrix} f_{k+1} - f_k - m_k h_k \\ m_{k+1} - m_k \end{pmatrix}$$

wobei h_k mit $\frac{2}{N}$ gegeben war. Der Quelltext sieht dann folgendermaßen aus:

```
runge = @(x) 1./(1+25*x.^2);
runge_abl = @(x) (-50*x)/(1+25*x^2)^2;
xreal = -1:0.01:1;

N = input('Anzahl der Stuetzstellen -1 :=N : ')

%Abstand Stuetzstellen h
h = 2./N;

%Stuetzstellen x
x = -1:h:1;

for i = 1:N+1
 %Stuetzwerte f
 f(i) = runge(x(i));
 %Ableitungen
 m(i) = runge_abl(x(i));
endfor

%Berechnung a_k, b_k nach 1.10
H = [h^3 , h^2 ; 3*h^2 , 2*h];
for i = 1:N
 r = H\[f(i+1)-f(i)-m(i)*h ; m(i+1)-m(i)];
 a(i) = r(1);
 b(i) = r(2);
 %c(i) = m(i)
 %d(i) = f(i)
endfor
```

```

%Interpolierende und Runge plotten auf Zerlegung M
M = 10 * N;
h_fein = 2/M;
x_fein = -1:h_fein:1;
k = 1;
for i=1:N
 %in jedem dieser Durchlaeufe ist der Spline-Abschnitt der Selbe
 for j=1:10
  s(k) = a(i)*(x_fein(k)-x(i))^3 +b(i)*(x_fein(k)-x(i))^2 +...
  m(i)*(x_fein(k)-x(i))+f(i);
  k = k + 1;
 endfor
endfor

s(k) = f(N+1);

figure(1);
plot(x_fein, runge(x_fein), "-;Funktion;", x_fein, ...
s,"-;Interpolation;")
```

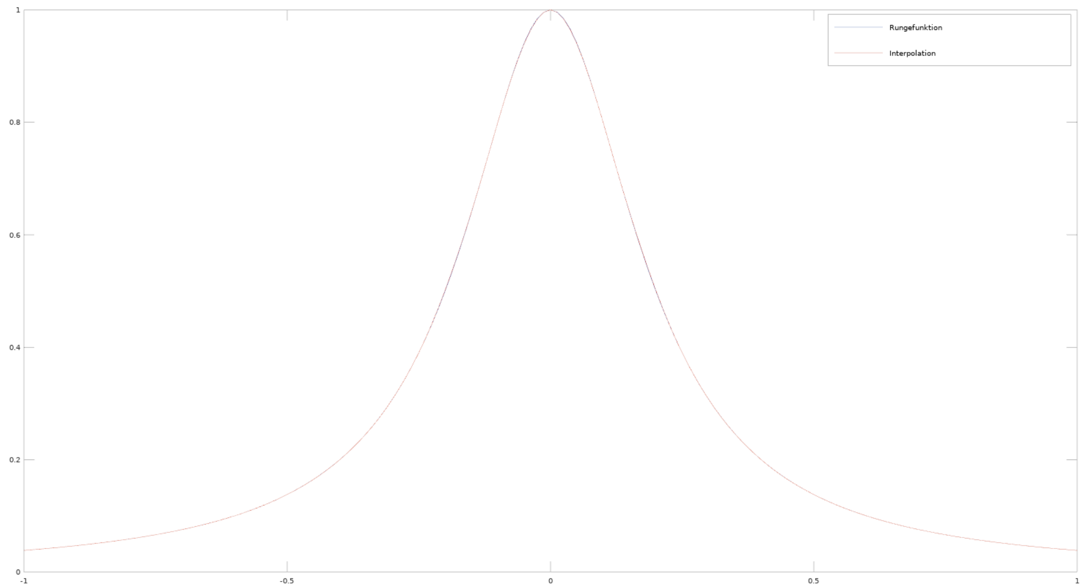

Abbildung 3: kubische Splineinterpolation mit $N = 16$

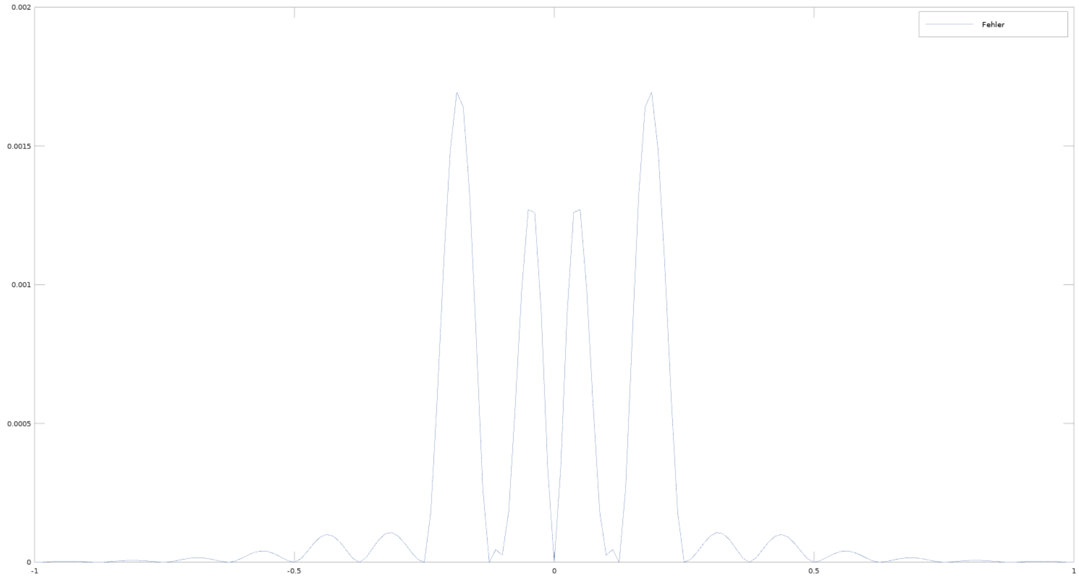

Abbildung 4: Fehler bei kubischer Splineinterpolation mit $N = 16$

1.2 Fehlerbetrachtung

Da Δ_M zehnmal so fein wie Δ_N ist, bedeutet das, dass man für jeden Spline den Fehler in 10 Punkten in seinem Definitionsbereich berechnet.

Bei linearer Interpolation kann man also deswegen den Fehler nach folgendem Muster ausrechnen:

$$\text{Fehler} = |f(x) - (n + \text{Abstand zur nächsten Stützstelle} \cdot m)|$$

wobei n und m zum jeweiligen Spline gehören und x die Werte in Δ_M durchläuft. Da die Fehlerfunktion laut Aufgabenstellung an den Stützstellen der Zerlegung Δ_M zu berechnen ist, lässt sich der nachfolgende Code auch für die Abschätzung des Fehlers (der auch an den Stützstellen von Δ_M gesucht ist) wiederverwenden. Der Quelltext dazu sieht folgendermaßen aus:

```
M = 10 * N
h_neu = 2/M
x_Fehler = -1:h_neu:1;

k = 1;
for i=1:N
 %in jedem dieser Durchlaufe ist der Spline-Abschnitt der Selbe
 for j=1:10
  y_Fehler(k) = abs(runge(x_Fehler(k)) - ...
   (n(i) + abs(abs(x_Fehler(k)) - abs(x(i))) * m(i)));
  k = k + 1;
 endfor
endfor

```

```
%Fehler an letzter Stuetzstelle ist 0
y_Fehler(k) = 0;

plot(x_Fehler, y_Fehler, "-; Fehler;")

% maximaler Fehler E
E = max(y_Fehler)
```

Für die Fehlerberechnung bei kubischer Interpolation haben wir wieder den Ansatz $s_k(x) = a_k(x - x_k)^3 + b_k(x - x_k)^2 + c_k(x - x_k) + d_k$ verwendet.

```
%Fehlerfunktion

k = 1;
 for i=1:N
 %in jedem dieser Durchlaeufe ist der Spline-Abschnitt der Selbe
 for j=1:10
  y_Fehler(k) = abs(runge(x_fein(k)) - ...
  (a(i)*(x_fein(k)-x(i))^3 +b(i)*(x_fein(k)-x(i))^2 +...
  m(i)*(x_fein(k)-x(i))+f(i)));
  k = k + 1;
 endfor
endfor

%Fehler an letzter Stuetzstelle ist 0
y_Fehler(k) = 0;

figure(2);
plot(x_fein, y_Fehler, "-; Fehler;")

%Maximaler Fehler
E = max(y_Fehler);
```

1.3 Diskussion der Ergebnisse

Der maximale Fehler $E(h_N)$ für $N = N_k = 4 \cdot 2^k$ mit $k = 0, ..., 4$ beträgt:

k	N_k	$E(h_{N_k})\ \mathcal{S}_1^0$	$E(h_{N_k})\ \mathcal{S}_3^1$
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

Man sieht also, dass bei großen N der Fehler sehr klein wird und die kubische Splineinterpolation besser als die lineare Interpolation ist.

Die exponentielle Konvergenzordnung ist

k	N_k	$EOC(h_{N_k}, h_{N_{k+1}})\ \mathcal{S}_1^0$	$EOC(h_{N_k}, h_{N_{k+1}})\ \mathcal{S}_3^1$
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

Mit $k = 11$, $N = 8192$ ist $h_{N_k} = \frac{2}{8192}$. Da EOC ein Maß für die Konvergenzgeschwindigkeit ist, bedeutet das, dass die kubische Spline-Interpolation doppelt so schnell gegen die Runge-Funktion konvergiert wie die lineare Interpolation.

2 Interpolation der anderen Funktion

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

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

2.1 Berechnung der Splines

Für Rechenvorschrift siehe 1.1

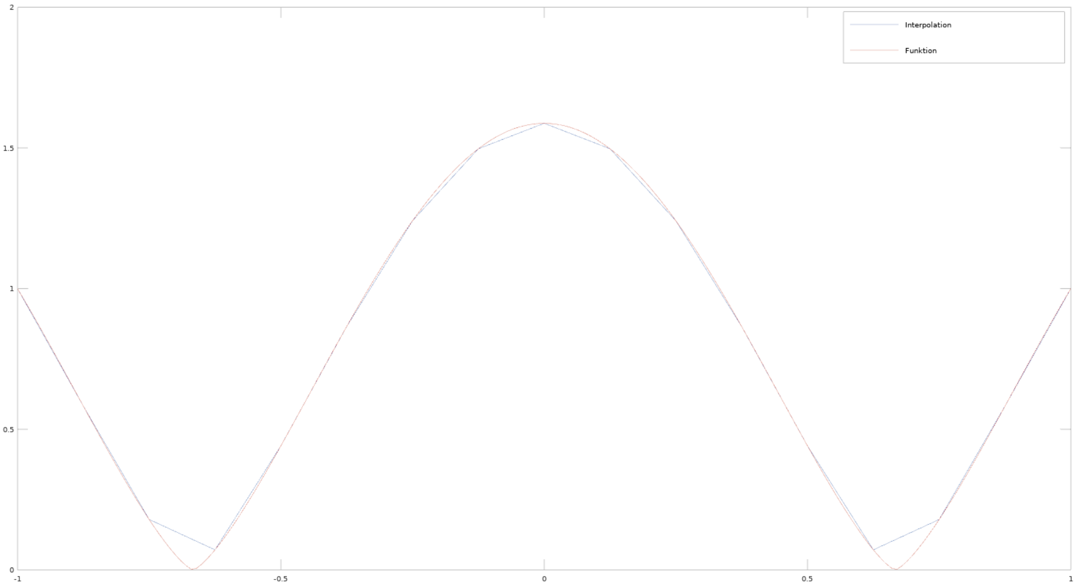

Abbildung 5: lineare Splineinterpolation mit $N = 16$

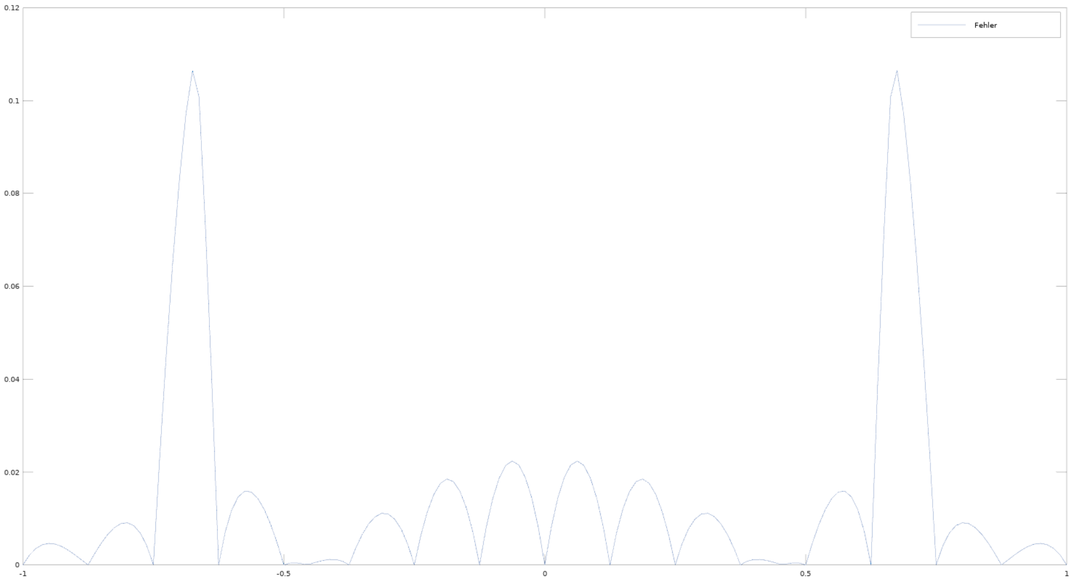

Abbildung 6: Fehler bei linearer Splineinterpolation mit $N = 16$

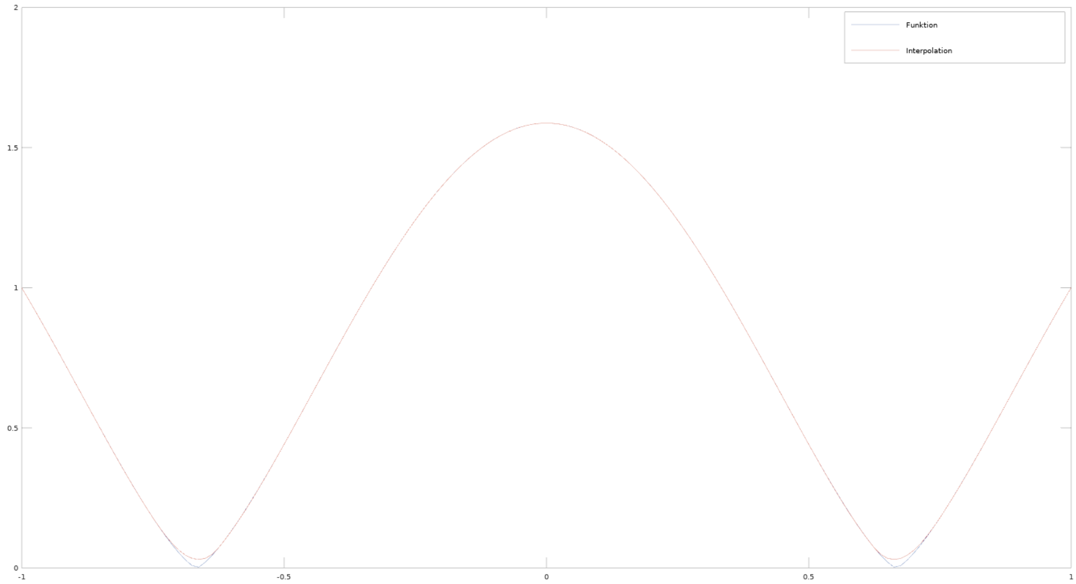

Abbildung 7: kubische Splineinterpolation mit $N = 16$

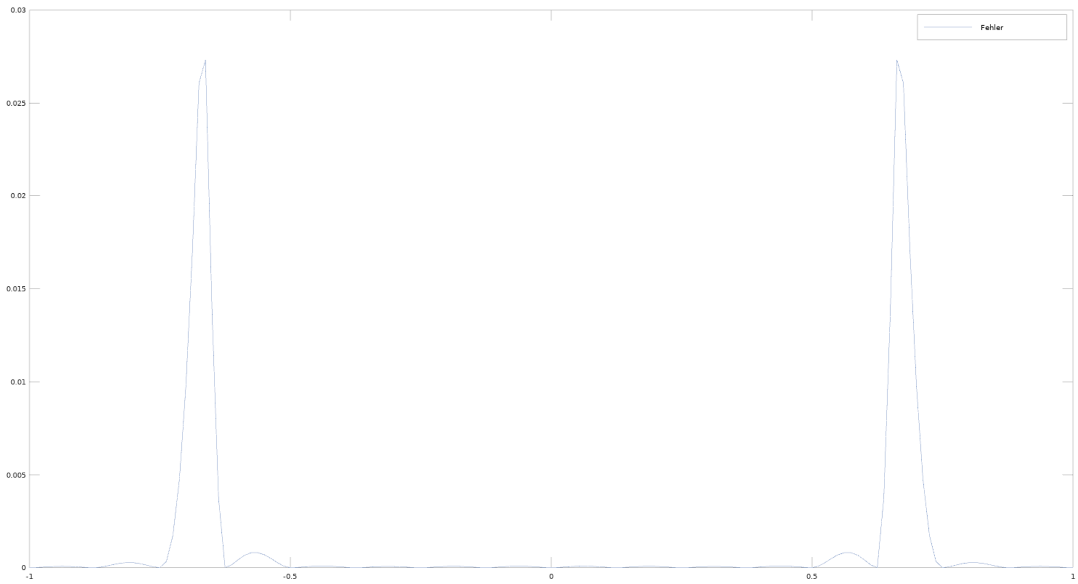

Abbildung 8: Fehler bei kubischer Splineinterpolation mit $N = 16$

2.2 Diskussion der Ergebnisse

Der maximale Fehler beträgt

k	N_k	$E(h_{N_k})\ \mathcal{S}_1^0$	$E(h_{N_k})\ \mathcal{S}_3^1$
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

Die exponentielle Konvergenzordnung ist

k	N_k	$EOC(h_{N_k}, h_{N_{k+1}})\ \mathcal{S}_1^0$	$EOC(h_{N_k}, h_{N_{k+1}})\ \mathcal{S}_3^1$
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

Mit $k = 11$, $N = 8192$ ist $h_{N_k} = \frac{2}{8192}$. Wenn EOC ein Maß für die Konvergenzgeschwindigkeit ist, dann konvergieren beide Ansätze - lineare und kubische Splieinterpolation - gleich schnell gegen diese Funktion.

Offensichtlich ist diese Funktion nicht so gut für eine Splineinterpolation geeignet wie die Runge-Funktion, weil die maximalen Fehler größer sind und die EOC kleiner ist.