Numerische Methoden der Physik

9 Interpolation, Extrapolation, Approximation

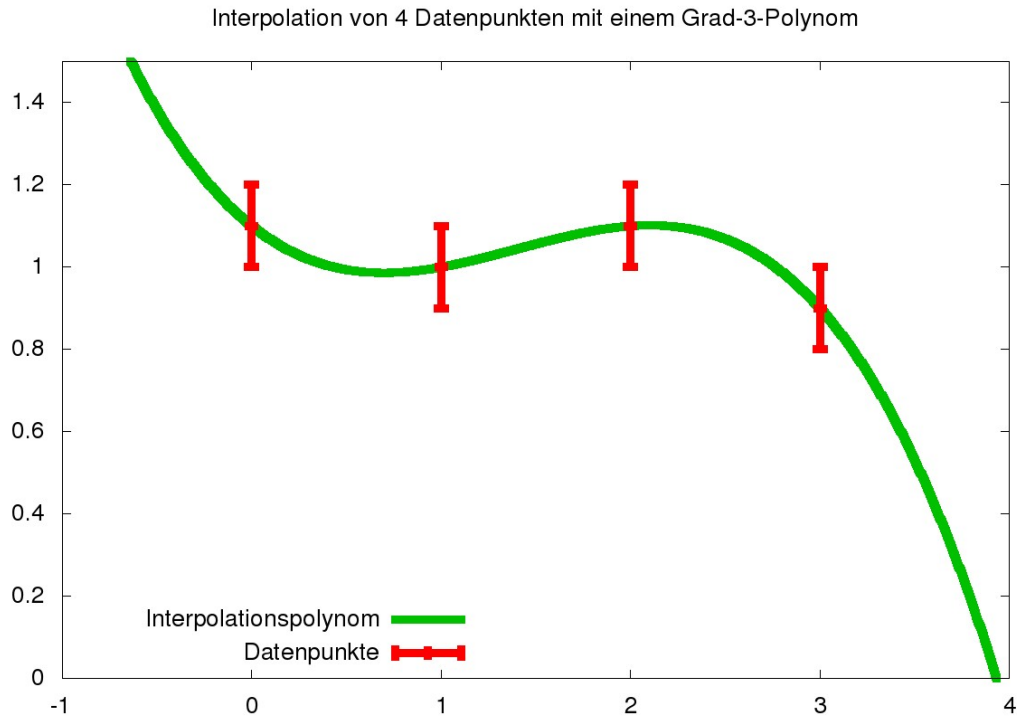
Marc Wagner

Institut für theoretische Physik
Johann Wolfgang Goethe-Universität Frankfurt am Main

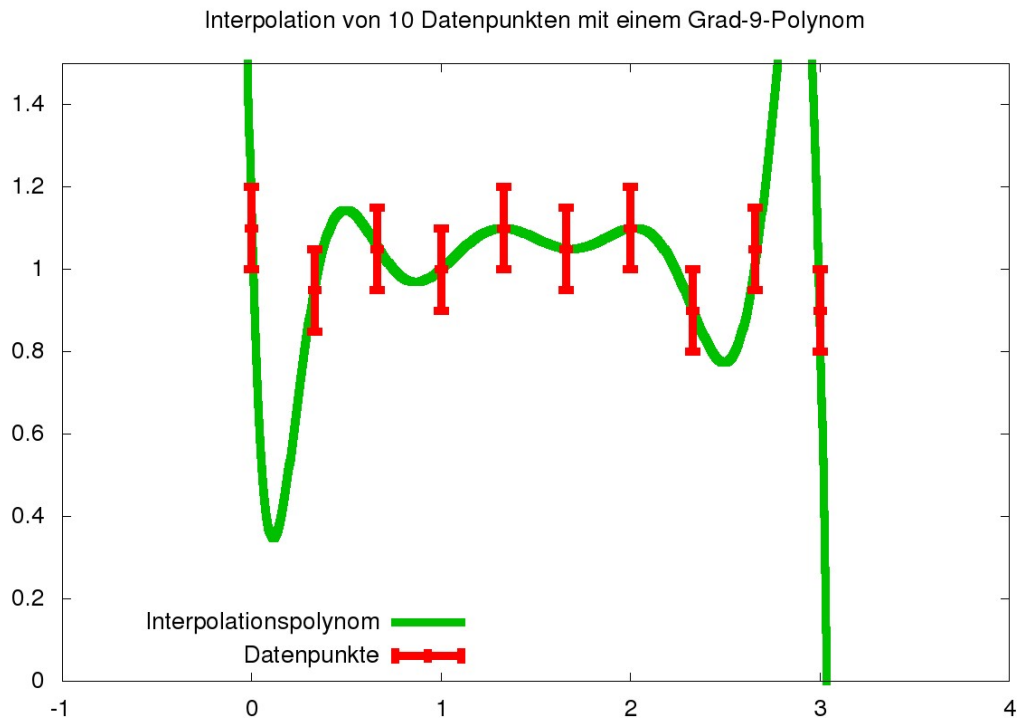
SS 2014

9.1 Polynominterpolation

- Vernünftig bei niedrigem Polynomgrad.



- **Starke Ozillationen bei hohem Polynomgrad!**


Interpolation von 10 Datenpunkten mit einem Grad-9-Polynom
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-1
0
1
2
3
4
Interpolationspolynom
Datenpunkte

9.2 Kubische Spline-Interpolation

- Kaum Oszillationen auch bei großer Menge an Datenpunkten.

```
1. // **********
2.
3.
4.
5. // spline.C
6.
7. // Kubische Spline-Interpolation (x_j und f_j vorgegeben).
8.
9. // Bestimmt die 2. Ableitungen mit Hilfe des Gauss-Algorithmus mit
10. // Rueckwaertssubstitution.
11.
12.
13.
14. // **********
15.
16.
17.
18. #include <math.h>
19. #include <stdio.h>
20. #include <stdlib.h>
21.
```

```
22.
23.
24. // **********
25.
26.
27.
28. #define __TEILPIVOTISIERUNG__
29.
30. // #define __SKALIERTE_TEILPIVOTISIERUNG__
31.
32.
33.
34. // **********
35.
36.
37.
38. // Dimension von A, b und x.
39. const int N = 10;
40.
41. // Matrix A (wird im Verlauf der Rechnung ueberschrieben).
42. double A[N][N];
43.
44. // Rechte-Seite-Vektor (wird im Verlauf der Rechnung ueberschrieben).
45. double b[N];
46.
47. // Ergebnis-Vektor.
48. double ddf[N];
```

```
49.
50. // Permutation der Zeilen aufgrund von Pivotisierung.
51. int p[N];
52.
53.
54. // *****
55.
56.
57. // Zu interpolierende Datenpunkte.
58.
59. double x[N] = {0.0, 0.33, 0.67, 1.0, 1.33, 1.67, 2.0, 2.33, 2.67, 3.0};
60. double f[N] = {1.1, 0.95, 1.05, 1.0, 1.1, 1.05, 1.1, 0.9, 1.05, 0.9};
61.
62.
63. // Anzahl der Sample-Punkte pro Teilstueck.
64. int num_samples = 21;
65.
66.
67. // **********
68.
69.
70.
71. void Print()
72. {
73.   int i1, i2;
74.
75.   for(i1 = 0; i1 < N; i1++)
```

```
    {
      for(i2 = 0; i2 < N; i2++)
        {
          fprintf(stderr, "%+5.2lf  ", A[p[i1]][i2]);
        }

      fprintf(stderr, "|  %+5.2lf\n", b[p[i1]]);
    }

  fprintf(stderr, "\n");
}



// **********


int main(int argc, char **argv)
{
  double d1, d2, d3;
  int i1, i2, i3;


  // *****
  // *****
  // *****
```

```
103.
104.   // Bestimme 2. Ableitungen (natuerliche Randbedingungen).
105.
106.   // *****
107.   // *****
108.   // *****
109.
110.
111.   // Generiere Matrix A und Vektor b zur Bestimmung der zweiten Ableitungen.
112.
113.   for(i1 = 0; i1 < N; i1++)
114.     {
115.       for(i2 = 0; i2 < N; i2++)
116.         A[i1][i2] = 0.0;
117.
118.       b[i1] = 0.0;
119.     }
120.
121.   // Natuerliche Randbedingungen.
122.
123.   A[0][0] = 1.0;
124.   b[0] = 0.0;
125.
126.   A[N-1][N-1] = 1.0;
127.   b[N-1] = 0.0;
128.
129.   for(i1 = 1; i1 < N-1; i1++)
```

```
130.     {
131.       A[i1][i1-1] = (x[i1  ]-x[i1-1]) / 6.0;
132.       A[i1][i1  ] = (x[i1+1]-x[i1-1]) / 3.0;
133.       A[i1][i1+1] = (x[i1+1]-x[i1  ]) / 6.0;
134.
135.       b[i1] =
136.         (f[i1+1]-f[i1]) / (x[i1+1]-x[i1]) - (f[i1]-f[i1-1]) / (x[i1]-x[i1-1]);
137.     }
138.
139.   // Intitialisiere Zeilenpermutation.
140.
141.   for(i1 = 0; i1 < N; i1++)
142.     p[i1] = i1;
143.
144.   Print();
145.
146.
147.   // *****
148.
149.
150.   // Spaltenausraeumen.
151.
152.   // Vorlesungsnotation:
153.   // i1 ~ n-1
154.   // i2 ~ j-1
155.   // i3 ~ k-1
156.
```

```
157. #ifdef __SKALIERTE_TEILPIVOTISIERUNG__
158. 
159.   // Maximum fuer jede Zeile von A speichern, bevor A durch Gauss-Algorithmus
160.   // ueberschrieben wird.
161. 
162.   double A_ij_max[N];
163. 
164.   for(i1 = 0; i1 < N; i1++)
165.     {
166.       A_ij_max[i1] = fabs(A[i1][0]);
167. 
168.       for(i2 = 1; i2 < N; i2++)
169.         {
170.           if(fabs(A[i1][i2]) > A_ij_max[i1])
171.             A_ij_max[i1] = fabs(A[i1][i2]);
172.         }
173.     }
174. 
175. #endif
176. 
177.   for(i1 = 0; i1 < N-1; i1++)
178.     // N-1 Spaltenausräumschritte.
179.     {
180.       // Finde beste Zeile gemaess Pivotstrategie.
181. 
182.       int index = i1;
183. 
```

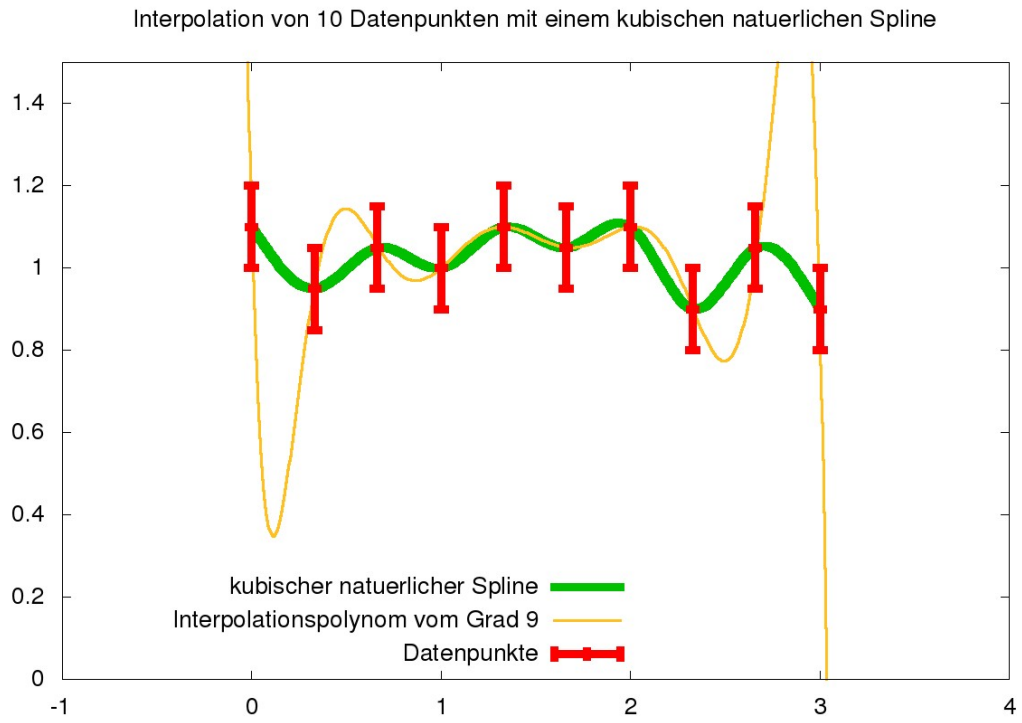
```
184. #ifdef __TEILPIVOTISIERUNG__
185.
186.       for(i2 = i1+1; i2 < N; i2++)
187.         {
188.           if(fabs(A[p[i2]][i1]) > fabs(A[p[i1]][i1]))
189.             index = i2;
190.         }
191.
192. #endif
193.
194. #ifdef __SKALIERTE_TEILPIVOTISIERUNG__
195.
196.       d1 = fabs(A[p[i1]][i1]) / A_ij_max[p[i1]];
197.
198.       for(i2 = i1+1; i2 < N; i2++)
199.         {
200.           d2 = fabs(A[p[i2]][i1]) / A_ij_max[p[i2]];
201.
202.           if(d2 > d1)
203.             index = i2;
204.         }
205.
206. #endif
207.
208.       i2 = p[i1];
209.       p[i1] = p[index];
210.       p[index] = i2;
```

```
211.
212.       // ***
213.
214.       for(i2 = i1+1; i2 < N; i2++)
215.         // Fuer alle verbleibenden Zeilen ...
216.         {
217.           d1 = A[p[i2]][i1] / A[p[i1]][i1];
218.
219.           // Zu Beginn jeder verbleibenden Zeile wird 0.0 "erzeugt".
220.           A[p[i2]][i1] = 0.0;
221.
222.           for(i3 = i1+1; i3 < N; i3++)
223.             {
224.               A[p[i2]][i3] -= d1 * A[p[i1]][i3];
225.             }
226.
227.           b[p[i2]] -= d1 * b[p[i1]];
228.         }
229.
230.       Print();
231.     }
232.
233.
234.   // *****
235.
236.
237.   // Rueckwaertssubstitution.
```

```
238.
239.   for(i1 = N-1; i1 >= 0; i1--)
240.     // Fuer alle Komponenten von x ...
241.     {
242.       ddf[i1] = b[p[i1]];
243.
244.       for(i2 = i1+1; i2 < N; i2++)
245.         {
246.           ddf[i1] -= A[p[i1]][i2] * ddf[i2];
247.         }
248.
249.       ddf[i1] /= A[p[i1]][i1];
250.     }
251.
252.   fprintf(stderr, "x = ( ");
253.
254.   for(i1 = 0; i1 < N-1; i1++)
255.     {
256.       fprintf(stderr, "%+5.2lf  ", ddf[i1]);
257.     }
258.
259.   fprintf(stderr, "%+5.2lf ).\n\n", ddf[N-1]);
260.
261.
262.   // *****
263.   // *****
264.   // *****
```

```
265.
266.   // Ausgabe des Splines.
267.
268.   // *****
269.   // *****
270.   // *****
271.
272.
273.   for(i1 = 0; i1 < N-1; i1++)
274.     {
275.       for(i2 = 0; i2 < num_samples; i2++)
276.         {
277.           double x_ = x[i1] + (x[i1+1]-x[i1])*(((double)i2) / ((double)(num_samples-1)));
278.
279.           double A_ = (x[i1+1] - x_) / (x[i1+1]-x[i1]);
280.           double B_ = (x_ - x[i1  ]) / (x[i1+1]-x[i1]);
281.
282.           double C_ = (pow(A_, 3.0) - A_) * pow(x[i1+1]-x[i1], 2.0) / 6.0;
283.           double D_ = (pow(B_, 3.0) - B_) * pow(x[i1+1]-x[i1], 2.0) / 6.0;
284.
285.           double y_ = A_*f[i1] + B_*f[i1+1] + C_*ddf[i1] + D_*ddf[i1+1];
286.
287.           fprintf(stdout, "%+.5e %+.5e\n", x_, y_);
288.         }
289.     }
290.
291.
```

```
292.   // *****
293.
294.
295.   return EXIT_SUCCESS;
296. }
297.
298.
299.
300. // **********
```


Interpolation von 10 Datenpunkten mit einem kubischen natuerlichen Spline
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-1
0
1
2
3
4
kubischer natuerlicher Spline
Interpolationspolynom vom Grad 9
Datenpunkte

9.3 Methode der kleinsten Fehlerquadrate

- Noch weniger Oszillationen bei Approximation mit einem einzigen Polynom niedrigen Grades.

```
1. // **********
2.
3.
4.
5. // least_squares.C
6.
7. // Polynomielle least-squares Approximation von Datenpunkten (x_j , f_j).
8.
9. // Loest das lineare Gleichungssystem A^T A a = A^T b mit Hilfe des
10. // Gauss-Algorithmus mit Rueckwaertssubstitution.
11.
12.
13.
14. // **********
15.
16.
17.
18. #include <math.h>
19. #include <stdio.h>
20. #include <stdlib.h>
```

```
21.
22.
23.
24. // **********
25.
26.
27.
28. #define __TEILPIVOTISIERUNG__
29.
30. // #define __SKALIERTE_TEILPIVOTISIERUNG__
31.
32.
33.
34. // **********
35.
36.
37.
38. // Dimension von A^T A, A^T b und a.
39. // const int N = 1;  // Grad-0-Polynom.
40. // const int N = 2;  // Grad-1-Polynom.
41. const int N = 3;  // Grad-2-Polynom.
42.
43. // Matrix A, eigentlich A^T A (wird im Verlauf der Rechnung ueberschrieben).
44. double A[N][N];
45.
46. // Rechte-Seite-Vektor b, eigentlich A^T b (wird im Verlauf der Rechnung
47. // ueberschrieben).
```

```
48. double b[N];
49. 
50. // Ergebnis-Vektor.
51. double a[N];
52. 
53. // Permutation der Zeilen aufgrund von Pivotisierung.
54. int p[N];
55. 
56. 
57. // *****
58. 
59. 
60. // Zu interpolierende Datenpunkte.
61. 
62. const int M = 10;
63. 
64. double x[M] = {0.0, 0.33, 0.67, 1.0, 1.33, 1.67, 2.0, 2.33, 2.67, 3.0};
65. double f[M] = {1.1, 0.95, 1.05, 1.0, 1.1, 1.05, 1.1, 0.9, 1.05, 0.9};
66. 
67. 
68. // **********
69. 
70. 
71. 
72. void Print()
73. {
74.   int i1, i2;
```

```
75.
76.   for(i1 = 0; i1 < N; i1++)
77.     {
78.       for(i2 = 0; i2 < N; i2++)
79.         {
80.           fprintf(stderr, "%+5.2lf  ", A[p[i1]][i2]);
81.         }
82.
83.       fprintf(stderr, "|  %+5.2lf\n", b[p[i1]]);
84.     }
85.
86.   fprintf(stderr, "\n");
87. }
88.
89.
90.
91. // **********
92.
93.
94.
95. int main(int argc, char **argv)
96. {
97.   double d1;
98.   int i1, i2, i3;
99.
100.
101.  // *****
```

```
102.
103.
104.   // Generiere Matrix A ~ A^T A und Vektor b ~ A^T b.
105.
106.   for(i1 = 0; i1 < N; i1++)
107.     {
108.       for(i2 = 0; i2 < N; i2++)
109.         {
110.           A[i1][i2] = 0.0;
111.
112.           for(i3 = 0; i3 < M; i3++)
113.             A[i1][i2] += pow(x[i3], (double)i1) * pow(x[i3], (double)i2);
114.         }
115.
116.       b[i1] = 0.0;
117.
118.       for(i2 = 0; i2 < M; i2++)
119.         b[i1] += pow(x[i2], (double)i1) * f[i2];
120.     }
121.
122.   // Intitialisiere Zeilenpermutation.
123.
124.   for(i1 = 0; i1 < N; i1++)
125.     p[i1] = i1;
126.
127.   Print();
128.
```

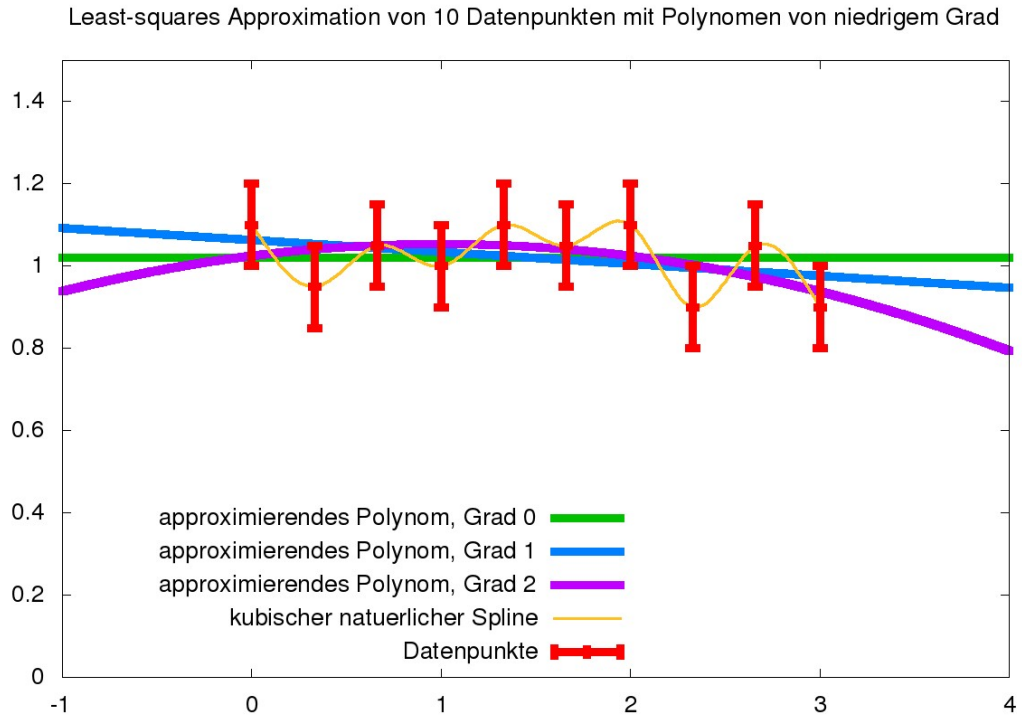
```
129.
130.   // *****
131.
132.
133.   // Spaltenausraeumen.
134.
135.   // Vorlesungsnotation:
136.   // i1 ~ n-1
137.   // i2 ~ j-1
138.   // i3 ~ k-1
139.
140. #ifdef __SKALIERTE_TEILPIVOTISIERUNG__
141.
142.   // Maximum fuer jede Zeile von A speichern, bevor A durch Gauss-Algorithmus
143.   // ueberschrieben wird.
144.
145.   double A_ij_max[N];
146.
147.   for(i1 = 0; i1 < N; i1++)
148.     {
149.       A_ij_max[i1] = fabs(A[i1][0]);
150.
151.       for(i2 = 1; i2 < N; i2++)
152.         {
153.           if(fabs(A[i1][i2]) > A_ij_max[i1])
154.             A_ij_max[i1] = fabs(A[i1][i2]);
155.         }
```

```
156.     }
157.
158. #endif
159.
160.   for(i1 = 0; i1 < N-1; i1++)
161.     // N-1 Spaltenausraeumschritte.
162.     {
163.       // Finde beste Zeile gemaess Pivotstrategie.
164.
165.       int index = i1;
166.
167. #ifdef __TEILPIVOTISIERUNG__
168.
169.       for(i2 = i1+1; i2 < N; i2++)
170.         {
171.           if(fabs(A[p[i2]][i1]) > fabs(A[p[i1]][i1]))
172.             index = i2;
173.         }
174.
175. #endif
176.
177. #ifdef __SKALIERTE_TEILPIVOTISIERUNG__
178.
179.       d1 = fabs(A[p[i1]][i1]) / A_ij_max[p[i1]];
180.
181.       for(i2 = i1+1; i2 < N; i2++)
182.         {
```

```
183.           d2 = fabs(A[p[i2]][i1]) / A_ij_max[p[i2]];
184.
185.           if(d2 > d1)
186.             index = i2;
187.         }
188.
189. #endif
190.
191.       i2 = p[i1];
192.       p[i1] = p[index];
193.       p[index] = i2;
194.
195.       // ***
196.
197.       for(i2 = i1+1; i2 < N; i2++)
198.         // Fuer alle verbleibenden Zeilen ...
199.         {
200.           d1 = A[p[i2]][i1] / A[p[i1]][i1];
201.
202.           // Zu Beginn jeder verbleibenden Zeile wird 0.0 "erzeugt".
203.           A[p[i2]][i1] = 0.0;
204.
205.           for(i3 = i1+1; i3 < N; i3++)
206.             {
207.               A[p[i2]][i3] -= d1 * A[p[i1]][i3];
208.             }
209.
```

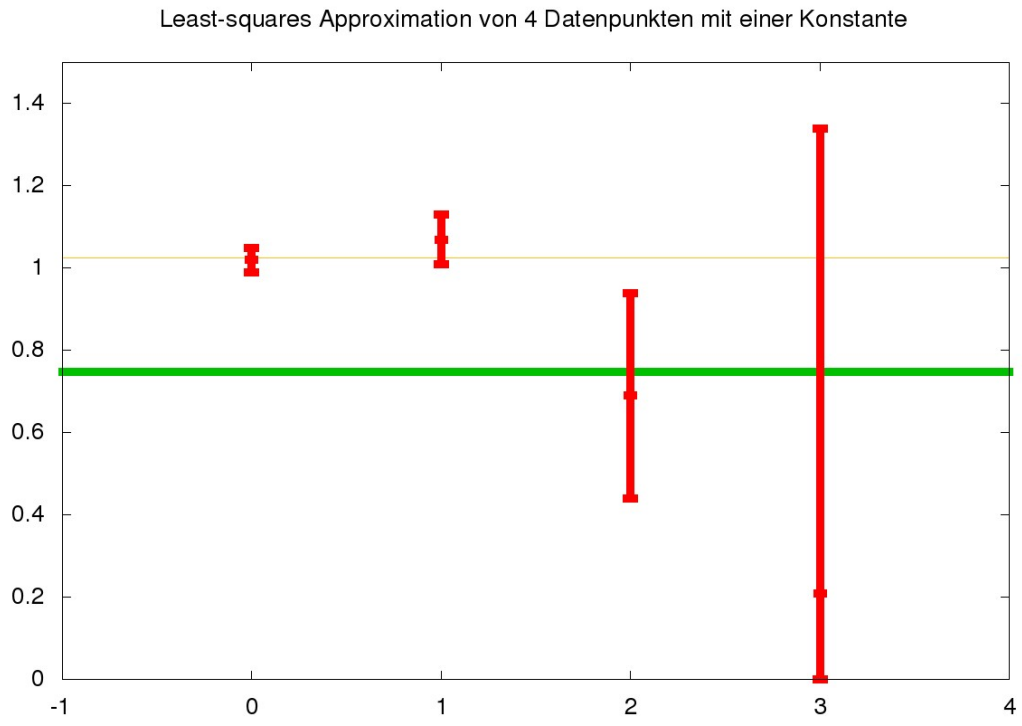
```
210.           b[p[i2]] -= d1 * b[p[i1]];
211.         }
212.
213.       Print();
214.     }
215.
216.
217.   // *****
218.
219.
220.   // Rueckwaertssubstitution.
221.
222.   for(i1 = N-1; i1 >= 0; i1--)
223.     // Fuer alle Komponenten von x ...
224.     {
225.       a[i1] = b[p[i1]];
226.
227.       for(i2 = i1+1; i2 < N; i2++)
228.         {
229.           a[i1] -= A[p[i1]][i2] * a[i2];
230.         }
231.
232.       a[i1] /= A[p[i1]][i1];
233.     }
234.
235.   fprintf(stderr, "a = ( ");
236.
```

```
237.   for(i1 = 0; i1 < N-1; i1++)
238.     {
239.       fprintf(stderr, "%+9.6lf  ", a[i1]);
240.     }
241.
242.   fprintf(stderr, "%+9.6lf ).\n\n", a[N-1]);
243.
244.
245.   // *****
246.
247.
248.   return EXIT_SUCCESS;
249. }
250.
251.
252.
253. // **********
```

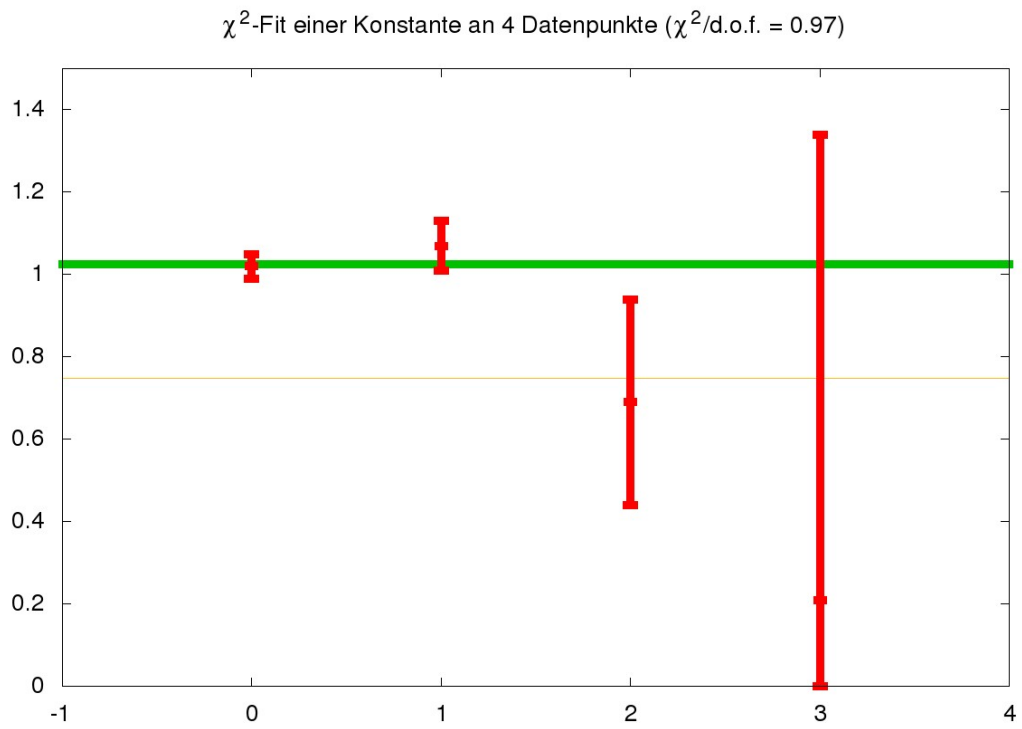

Least-squares Approximation von 10 Datenpunkten mit Polynomen von niedrigem Grad
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-1
0
1
2
3
4
approximierendes Polynom, Grad 0
approximierendes Polynom, Grad 1
approximierendes Polynom, Grad 2
kubischer natuerlicher Spline
Datenpunkte

9.4 χ^2-Fitting

- **Least-squares Fit: Fehlerbalken gehen nicht in den Fit ein!**



- χ^2-Fit: Fehlerbalken gehen in den Fit ein, Datenpunkte mit kleinen Fehlern erhalten großes Gewicht.


χ^2-Fit einer Konstante an 4 Datenpunkte (χ^2/d.o.f. = 0.97)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-1
0
1
2
3
4