



Musterlösung Übungsserie 3 Mathematik für Informatiker III

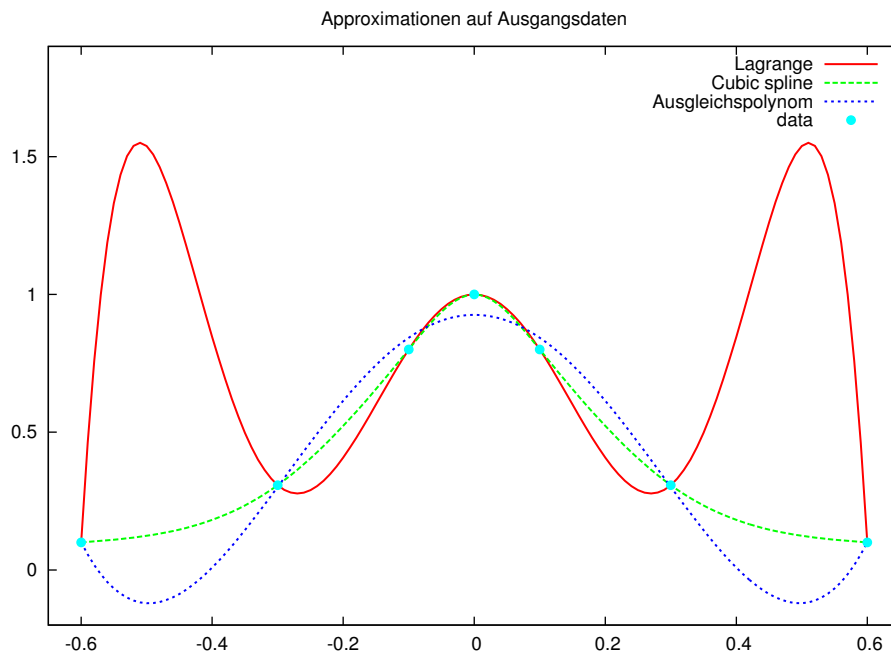
Aufgabe 1: Interpolation – Lagrange-Polynom, Kubische Splines und Ausgleichsprobleme

Berechne zu den Wertepaaren (x_i, y_i) , $i = 0, \dots, 6$, mit den Stützstellen $x_{0..6} = (-.6, -.3, -.1, 0, .1, .3, .6)$ und den Daten- oder Meßwerten $y_i = \frac{1}{1+25x_i^2}$ jeweils

- (i) das interpolierende Polynom (Lagrange, 1.Semester)
- (ii) den natürlichen kubischen Spline
- (iii) ein Ausgleichspolynom 4.Ordnung (Gauß mit Ansatz $u(x) = ax^4 + bx^3 + cx^2 + dx + e$)

Stelle die erhaltenen Approximationen grafisch dar.

Lösung:



Man sieht leicht, dass das Lagrange - Polynom stark schwingt, aber durch jeden Meßwert läuft.

Das Ausgleichspolynom 4.Ordnung wird nicht durch alle Meßwerte gezwungen und schwingt deutlich weniger als das Lagrange-Polynom.

Der kubische Spline geht direkt durch alle Meßwerte ohne stark zu schwingen.

Gib das tridiagonale Gleichungssystem des kubischen Splines mit dem zugehörigen Lösungsvektor der $z = (z_1, \dots, z_{n-1})^T$ an.

$$\begin{pmatrix} 1 & 0.2 & & & \\ 0.2 & 0.6 & 0.1 & & \\ & 0.1 & 0.4 & 0.1 & \\ & & 0.1 & 0.6 & 0.2 \\ & & & 0.2 & 1 \end{pmatrix} z = \begin{pmatrix} 10.615385 \\ -2.769231 \\ -24.000000 \\ -2.769231 \\ 10.615385 \end{pmatrix}$$

$$z = (10.180995, 2.171946, -61.085973, 2.171946, 10.180995)^T$$

Zur Kontrolle:

Das System nach der Vorwärtselimination:

$$\begin{pmatrix} 1 & 0.2 & & & \\ 0 & 0.56 & 0.1 & & \\ & 0 & 0.382143 & 0.1 & \\ & & 0 & 0.573832 & 0.2 \\ & & & 0 & 0.930293 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 2.181459 \\ 2.061216 \end{pmatrix} = \begin{pmatrix} 10.615385 \\ -4.892308 \\ -23.126374 \\ 3.282531 \\ 9.471310 \end{pmatrix}$$

Das System nach der Rückwärtselimination:

$$\begin{pmatrix} 1 & 0.2 & & & \\ 0 & 0.56 & 0.1 & & \\ & 0 & 0.382143 & 0.1 & \\ & & 0 & 0.573832 & 0.2 \\ & & & 0 & 0.930293 \end{pmatrix} \begin{pmatrix} 10.180995 \\ 2.171946 \\ -61.085973 \\ 2.171946 \\ 10.180995 \end{pmatrix} = \begin{pmatrix} 10.615385 \\ -4.892308 \\ -23.126374 \\ 3.282531 \\ 9.471310 \end{pmatrix}$$

Erstelle eine Tabelle der Koeffizienten (a_i, b_i, c_i, d_i) , $i = 1, \dots, 6$, des kubischen Splines.

i	a	b	c	d
1	5.656109	0.000000	0.183258	0.100000
2	-6.674208	5.090498	1.710407	0.307692
3	-105.429864	1.085973	2.945701	0.800000
4	105.429864	-30.542986	0.000000	1.000000
5	6.674208	1.085973	-2.945701	0.800000
6	-5.656109	5.090498	-1.710407	0.307692

Gib die Koeffizienten des Ausgleichspolynoms an.

$$z = (0.925975, 0.000000, -8.486668, 0.000000, 17.203533)^T$$

Es fällt sofort auf, daß die Koeffizienten der geraden Potenzen im Polynom identisch Null sind. Der Grund dafür ist die Symmetrie der Stützstellen und Meßwerte.

Zur Kontrolle:

Matrix A:

$$A = \begin{pmatrix} 1.000000 & -0.600000 & 0.360000 & -0.216000 & 0.129600 \\ 1.000000 & -0.300000 & 0.090000 & -0.027000 & 0.008100 \\ 1.000000 & -0.100000 & 0.010000 & -0.001000 & 0.000100 \\ 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 1.000000 & 0.100000 & 0.010000 & 0.001000 & 0.000100 \\ 1.000000 & 0.300000 & 0.090000 & 0.027000 & 0.008100 \\ 1.000000 & 0.600000 & 0.360000 & 0.216000 & 0.129600 \end{pmatrix}$$

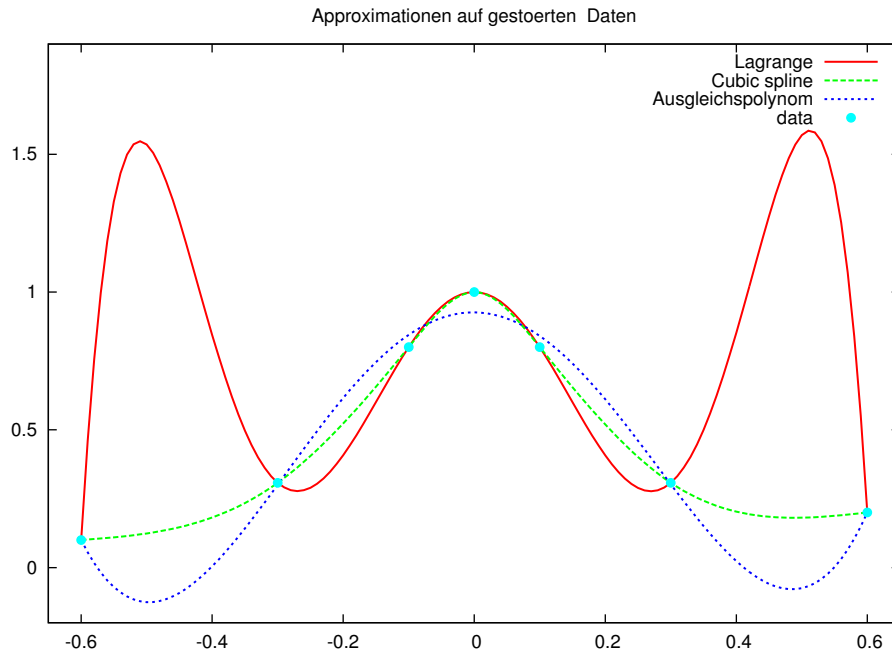
Zu lösendes System $A^T A z = A^T y$

$$\begin{pmatrix} 7.000000 & 0 & 0.920000 & 0 & 0.275600 \\ 0 & 0.920000 & 0 & 0.275600 & 0 \\ 0.920000 & 0 & 0.275600 & 0 & 0.094772 \\ 0 & 0.275600 & 0 & 0.094772 & 0 \\ 0.275600 & 0 & 0.094772 & 0 & 0.033724 \end{pmatrix} z = \begin{pmatrix} 3.415385 \\ 0 \\ 0.143385 \\ 0 \\ 0.031065 \end{pmatrix}$$

Gauß ohne Pivotierung liefert:

$$\begin{pmatrix} 7.000000 & 0 & 0.920000 & 0 & 0.275600 \\ 0 & 0.920000 & 0 & 0.275600 & 0 \\ 0 & 0 & 0.154686 & 0 & 0.058550 \\ 0 & 0 & 0 & 0.012212 & 0 \\ 0 & 0 & 0 & 0 & 0.000711 \end{pmatrix} z = \begin{pmatrix} 3.415385 \\ 0 \\ -0.305495 \\ 0 \\ 0.012229 \end{pmatrix}$$

Störe nun den Meßwert y_6 durch Addition von 0.1 und berechne zu diesen gestörten Daten wiederum Lagrange-Polynom, kubischen Spline und Ausgleichspolynom der Ordnung 4. Visualisiere die Approximationen!



Untersuche die Auswirkung der Störung auf die drei Approximationen. Erstelle dazu eine Tabelle mit den Werten der Approximationen mit den nicht gestörten Daten (1.Spalte), den gestörten Daten (2.Spalte) und dem Betrag der Differenzen beider an den Zwischenstellen $\{-.45, -.2, -.05, .05, .2, .45\}$ (3.Spalte). Interpretiere die Reaktion der Approximationen auf die Störung der Daten!

Tabelle Lagrange:

x	original	gestört	Differenz
-0.450000	1.260427	1.258278	0.002148
-0.200000	0.407692	0.407869	0.000176
-0.050000	0.946484	0.946458	0.000027
0.050000	0.946484	0.946516	0.000031
0.200000	0.407692	0.407340	0.000353
0.450000	1.260427	1.275466	0.015039

Tabelle Ausgleichspolynom:

x	original	gestört	Differenz
-0.450000	-0.087123	-0.091906	0.004783
-0.200000	0.614034	0.615529	0.001495
-0.050000	0.904866	0.906177	0.001311
0.050000	0.904866	0.903878	0.000988
0.200000	0.614034	0.610786	0.003248
0.450000	-0.087123	-0.059150	0.027973

Tabelle natürlicher kubischer Spline:

x	original	gestört	Differenz
-0.450000	0.146578	0.146539	0.000039
-0.200000	0.522964	0.523034	0.000070
-0.050000	0.936821	0.936721	0.000101
0.050000	0.936821	0.937167	0.000346
0.200000	0.522964	0.519462	0.003501
0.450000	0.146578	0.184485	0.037907

Das Lagrange - Polynom weist starke Änderungen in den beiden außen gelegenen Zwischenstellen obwohl nur am rechten Rand gestört wurde. Das ist nicht weiter verwunderlich, denn das sind ja gerade die Bereiche der stärksten Schwingungen. Interessanterweise zeigt das Lagrange - Polynom in der Summe über alle Differenzen die kleinste absolute Änderung auf die Störung am rechten Rand.

Das Ausgleichspolynom 4.Ordnung reagiert auf die Störung auch an den Rändern am stärksten, allerdings sind an allen anderen Zwischenstellen zumindest kleinere Änderungen zu beobachten. Obwohl das Ausgleichspolynom im Vergleich zum Lagrange - Polynom deutlich schwächer schwingt, ist die Summe der absoluten Änderungen hier größer.

Der natürliche kubische Spline reagiert in der Summe am stärksten auf die Störung des rechten Randes, allerdings kann man hier von einer eindeutig lokalen Reaktion sprechen: Je weiter man sich vom Ort der Störung entfernt, desto schwächer wird auch ihr Einfluss. Die linke Zwischenstelle reagiert nur noch mit einem Promille der Änderung am rechten Punkt.

Zusatzaufgabe A:

Modifiziere die Gaußelimination so, dass das bei der Interpolation mit natürlichen kubischen Splines entstehende lineare symmetrische tridiagonale Gleichungssystem mit n Gleichungen in n Unbekannten mit maximal n Divisionen gelöst wird!

Siehe Quelltext.

```

1 #include <stdio.h>
2
3 #define Ypertubation .1
4
5 #define N 7
6 #define NN 5
7
8 // data
9 double x[] = { -.6, -.3, -.1, 0, .1, .3, .6 };
10 double xxx[] = { -.45, -.2, -.05, .05, .2, .45 };
11 double y[N], y1[N];
12
13
14 // natural cubic spline stuff
15 typedef struct { double a, b, c, d; } c_cspl; // one piece of a spline
16 c_cspl cspl[N-1], cspl1[N-1]; //coefficients of complete splines
17
18 double cubic_spline( double xx, c_cspl cspl[N-1])
19 { // compute value of spline CSPL on point XX
20     int i;
21     if (( xx < x[0] ) || ( xx > x[N-1]+1e-14 ) )
22         return 0;
23     i = 0; // will held coefficients number
24     while ( ( i < N-1 ) && ( xx + 1e-14 > x[i+1] ) ) i++;
25     if ( i >= N-1 ) i--;
26     return ((cspl[i].a * (xx-x[i]) + cspl[i].b) * (xx-x[i])
27             + cspl[i].c) * (xx-x[i]) + cspl[i].d;
28 }
29
30
31 void cspline_coeff( double x[N], double y[N], c_cspl cspl[NN] )
32 { // build natural cubic spline through (x_i, y_i)
33     // note: lin.system dimension is N-2
34     double dd[N-2], ddd[N-2]; //2 diags
35     double rr[N-2]; // right hand side

```

```

36 double zz[N]; // solution of linear system, index shifted
37 double divsaver[N-2], tmp;
38 int i, j;
39
40 // build system to solve
41 dd[0] = 2*(x[2]-x[0]); // alpha
42 ddd[0] = x[2]-x[1]; // beta
43 rr[0] = 6*((y[2]-y[1])/(x[2]-x[1]))
44 - ((y[1]-y[0])/(x[1]-x[0]));
45
46 // index shift upper bound in next loop:
47 // N-1 from system definition
48 // N-2 from C array addressing
49 // N-3 from last complete row in system
50 for ( i=1; i < N-3; i++ ) {
51 dd[i] = 2*(x[i+2]-x[i]); // alpha
52 ddd[i] = x[i+2]-x[i+1]; // beta
53 rr[i] = 6*((y[i+2]-y[i+1])/(x[i+2]-x[i+1]))
54 - ((y[i+1]-y[i+0])/(x[i+1]-x[i+0]));
55 }
56 dd[N-3] = 2*(x[N-1]-x[N-3]); // alpha
57 ddd[N-3] = 0;
58 rr[N-3] = 6*((y[N-1]-y[N-2])/(x[N-1]-x[N-2]))
59 - ((y[N-2]-y[N-3])/(x[N-2]-x[N-3]));
60
61 // special symmetric tridiag solver
62 for ( i=0; i < N-3; i++ ) {
63 divsaver[i]=1.0/dd[i];
64 dd[i+1] -= (ddd[i]*ddd[i] * divsaver[i]);
65 rr[i+1] -= (rr[i]*ddd[i] * divsaver[i]);
66 }
67 divsaver[N-3] = 1.0/dd[N-3];
68
69 // backward substitution
70 zz[N-1] = 0; //natural spline !!!
71 for ( i=N-2; i > 0; ) {
72 i--;
73 tmp = 0;
74 if ( i < N-3 ) tmp = ddd[i] * zz[i+2];
75 zz[i+1] = (rr[i] - tmp) * divsaver[i];
76 }
77 zz[0] = 0; //natural spline !!!
78
79 printf("zz=");
80 for ( i=1; i < N-1; i++) printf("%10lf", zz[i] );
81 printf("\n");
82
83 // compute coefficients a b c d
84 // numbering 0..N-1, so i-1 on right hand side is just i
85 for ( i=1; i < N; i++ ) {
86 cspl[i-1].a = (zz[i] - zz[i-1]) /
87 ( 6.0*(x[i]-x[i-1]));
88 cspl[i-1].b = zz[i-1] / 2.0;
89 cspl[i-1].c = (y[i]-y[i-1])/(x[i]-x[i-1])
90 - (zz[i]+2*zz[i-1])*(x[i]-x[i-1])/6.0;
91 cspl[i-1].d = y[i-1];
92 }
93
94 printf("%3c_%12c_%12c_%12c_%12c\n", 'i', 'a', 'b', 'c', 'd');
95 for ( i=0; i < N-1; i++ )
96 printf("%3d_%12lf_%12lf_%12lf_%12lf\n",

```

```

97         i+1, cspl[i].a, cspl[i].b, cspl[i].c, cspl[i].d);
98     }
99
100
101 // least squares coefficients
102 double z[NN], z1[NN];
103
104
105 double gauss_polynom( double xx, double z[NN])
106 { // compute value of Gaussian polynomial with coefficients Z at point XX
107     int i, n;
108     n = NN; // n number of ansatz functions
109     double yy = z[n-1]; // do it ala Horner
110     for ( i = n-2; i >= 0; i-- ) yy = yy * xx + z[i];
111     return yy;
112 }
113
114
115 void gauss_coeff( double x[N], double y[N], double z[NN] )
116 { // compute coefficients Z of gaussian polynomial
117     double A[N][NN], AtA[NN][NN], AtY[NN];
118     int m, n, i, j, k, l;
119     double lambda;
120     // set up dimension
121     m = N; // m number of data pairs
122     n = NN; // n number of ansatz functions
123     // build matrix A
124     for( i=0; i < m; i++ ) {
125         A[i][0] = 1;
126         for ( j=1; j < n; j++ ) A[i][j] = A[i][j-1] * x[i];
127     }
128
129     // build A^T A
130     for ( k = 0; k < n; k++ ) {
131         for ( l = 0; l < n; l++ ) {
132             AtA[k][l] = 0;
133             for ( i = 0; i < m; i++ ) AtA[k][l] += (A[i][k] * A[i][l]);
134         }
135     }
136
137     // build A^T y
138     for ( k = 0; k < n; k++ ) {
139         AtY[k] = 0;
140         for ( i = 0; i < m; i++ )
141             AtY[k] += A[i][k] * y[i];
142     }
143
144     // gauss elimination without pivoting
145     for ( k = 0; k < n; k++ ) {
146         for ( l = k+1; l < n; l++ ) {
147             lambda = AtA[l][k] / AtA[k][k];
148             for ( i=k+1; i < n; i++ ) AtA[l][i] -= (lambda * AtA[k][i]);
149             AtY[l] -= (lambda * AtY[k]);
150             AtA[l][k] = 0;
151         }
152     }
153
154     // back solve
155     for ( i=n; i > 0; ) {
156         i--;
157         lambda = 0;

```

```

158     for ( j=i+1; j < n; j++ ) lambda += AtA[i][j] * z[j];
159     z[i] = ( AtY[i] - lambda ) / AtA [i][i];
160 }
161
162 printf("  z = ");
163 for ( i=0; i < n; i++) printf(" %10lf", z[i] );
164 printf("\n");
165 }
166
167
168 double lagrange( double xx, double x[], double y[] )
169 { // compute value of Lagrangian polynomial at point XX
170     int i, j;
171     double yy = 0, z, n;
172
173     for ( i = 0; i < N; i++ ) {
174         z = n = 1;
175         for ( j=0; j < N; j++ )
176             if ( i != j )
177                 { z *= (xx - x[j]); n *= (x[i] - x[j]); }
178         yy += y[i] * z / n;
179     }
180     return yy;
181 }
182
183
184 int main(void) {
185     int i;
186     double xx, A, B;
187
188     // data setup
189     for ( i=0; i<N; i++ ) y[i] = 1.0 / (1+25*x[i]*x[i]);
190     for ( i=0; i<N; i++ ) y1[i] = y[i];
191     y1[N-1] += Ypertubation;
192
193     printf(" DATA_X"); for ( i=0; i<N; i++ ) printf(" %10lf",x[i]);
194     printf("\n DATA_Y"); for ( i=0; i<N; i++ ) printf(" %10lf",y[i]);
195
196     printf("\n\nCUBIC_SPLINE... original_data\n");
197     cspline_coeff( x, y, cspl );
198     printf("\n\nCUBIC_SPLINE... pertubated_data\n");
199     cspline_coeff( x, y1, cspl1 );
200
201     printf("\n\nGAUSS_Least_squares... original_data\n");
202     gauss_coeff( x, y, z );
203     printf("\n\nGAUSS_Least_squares... pertubated_data\n");
204     gauss_coeff( x, y1, z1 );
205
206
207     printf("\n\n Table_Lagrange_on_midpoints\n");
208     printf(" %10s %10s %10s %10s\n", "x", "orig", "pert", "diff");
209     for ( i=0; i < N-1; i++ ) {
210         A = lagrange(xxx[i],x,y);
211         B = lagrange(xxx[i],x,y1);
212         printf(" %10lf %10lf %10lf %10lf\n",
213             xxx[i], A, B, fabs(A-B));
214     }
215     /*
216     tables of gaussian and cubic spline .... SKIPPED
217     */
218 } // main

```

Aufgabe 2: Numerische Integration

Berechne in doppelter Genauigkeit Näherungen für den Wert der Integrale

$$\int_0^{\pi/2} e^{2x} \cos(x) dx \quad \text{und} \quad \int_0^1 \sqrt{x^3} dx$$

mit

- (i) der summierten Trapezregel
- (ii) der Simpsonregel
- (iii) dem Romberg - Verfahren.

Beginne jeweils mit der Schrittweite $h_1 = \frac{b-a}{2}$ und setze mit halbiertes Schrittweite $h_k = \frac{b-a}{2^k} = h_{k-1}/2$, $k = 1, 2, \dots$ fort (das ist die schon bekannte *Romberg-Schrittweitenfolge*). Verwende als Abbruchkriterium bei Trapez- und Simpson-Regel $|T_{2^k} - T_{2^{k-1}}| \leq \delta |T_2|$ bzw. $|S_{2^k} - S_{2^{k-1}}| \leq \delta |S_2|$ mit $\delta = 10^{-12}$. Terminiere das Romberg-Verfahren, wenn zum ersten Mal $|R_k^k - R_k^{k-1}| \leq \delta |R_k^0|$ erfüllt ist.

Erstelle für beide Integranden eine Tabelle, in der für jedes $k = 1, 2, \dots$ die berechneten Werte T_{2^k} , S_{2^k} , R_k^0 , R_k^1 , \dots , R_k^k angegeben werden. (Natürlich nur die Werte, die auch wirklich berechnet wurden! Wenn die Simpson-Regel beispielsweise bei $k = 3$ terminiert, dann stehen in der Spalte von S_{2^k} eben nur drei Werte.)

Werte für $\int_0^{\pi/2} e^{2x} \cos(x) dx$, $\delta = 10^{-12}$, exakter Wert = 4.2281385								
k	Trapez	Simpson	Romberg					
1	3.0642476	3.8238640	3.9189968					
2	3.9134172	4.1964737	4.1964737	4.2149722				
3	4.1478953	4.2260547	4.2260547	4.2280268	4.2282340			
4	4.2079788	4.2280067	4.2280067	4.2281368	4.2281385	4.2281382		
5	4.2230924	4.2281303	4.2281303	4.2281385	4.2281385	4.2281385	4.2281385	
6	4.2268766	4.2281380	4.2281380	4.2281385	4.2281385	4.2281385	4.2281385	4.2281385
7	4.2278230	4.2281385						
8	4.2280596	4.2281385						
9	4.2281188	4.2281385						
10	4.2281336	4.2281385						
11	4.2281373	4.2281385						
12	4.2281382	4.2281385						
13	4.2281384							
14	4.2281385							
15	4.2281385							
16	4.2281385							
17	4.2281385							
18	4.2281385							
19	4.2281385							
20	4.2281385							
21	4.2281385							

Alle drei Verfahren berechnen den exakten Wert des Integrales mit der geforderten Genauigkeit. Dabei brauchen Trapez-Regel ca. viermal und Simpson-Regel doppelt so viele Schrittweitenverfeinerungen wie Romberg.

Werte für $\int_0^1 \sqrt{x^3} dx$ $\delta = 10^{-12}$, exakter Wert = 0.4000000								
k	Trapez	Simpson	Romberg					
1	0.4267767	0.4023689	0.4023689					
2	0.4070181	0.4004319	0.4004319	0.4003028				
3	0.4018125	0.4000772	0.4000772	0.4000536	0.4000496			
4	0.4004634	0.4000137	0.4000137	0.4000095	0.4000088	0.4000086		
5	0.4001177	0.4000024	0.4000024	0.4000017	0.4000016	0.4000015	0.4000015	
6	0.4000297	0.4000004	0.4000004	0.4000003	0.4000003	0.4000003	0.4000003	0.4000003
7	0.4000075	0.4000001	0.4000001	0.4000001	0.4000000	0.4000000	0.4000000	...
8	0.4000019	0.4000000	0.4000000	0.4000000	0.4000000	0.4000000	0.4000000	...
9	0.4000005	0.4000000	0.4000000	0.4000000	0.4000000	0.4000000	0.4000000	...
10	0.4000001	0.4000000						
11	0.4000000	0.4000000						
12	0.4000000	0.4000000						
13	0.4000000	0.4000000						
14	0.4000000	0.4000000						
15	0.4000000	0.4000000						
16	0.4000000							
17	0.4000000							
18	0.4000000							
19	0.4000000							
20	0.4000000							

Wieder gelingt es allen drei Verfahren, die Abbruchbedingung zu erfüllen und den exakten Wert zu berechnen. Allerdings benötigen sowohl Simpson als auch Romberg diesmal eine deutlich kleinere Schrittweite (Faktor 1/8). Weitere Aussagen kann man aus den berechneten Näherungswerten leider nicht ablesen. Da helfen die nächsten beiden Tabellen weiter.

Gib für jeden Integranden in einer zweiten Tabelle den Fehler zum exakten Integral an! (Hinweis: $\int_0^{\pi/2} e^{2x} \cos(x) dx = \frac{e^{\pi}-2}{5}$, $\sqrt{x^3}$ bitte selbst integrieren!). Vergleiche beide Tabellen. Interpretiere die beobachteten Unterschiede! Versuche eine Begründung für das Verhalten des Romberg-Verfahrens beim zweiten Integranden zu geben!

Fehler für $\int_0^{\pi/2} e^{2x} \cos(x) dx$								
k	Trapez	Simpson	Romberg					
1	1e+00	4e-01	3e-01					
2	3e-01	3e-02	3e-02	1e-02				
3	8e-02	2e-03	2e-03	1e-04	1e-04			
4	2e-02	1e-04	1e-04	2e-06	1e-08	4e-07		
5	5e-03	8e-06	8e-06	3e-08	9e-11	5e-11	4e-10	
6	1e-03	5e-07	5e-07	4e-10	4e-13	5e-14	2e-15	1e-13
7	3e-04	3e-08						
8	8e-05	2e-09						
9	2e-05	1e-10						
10	5e-06	8e-12						
11	1e-06	5e-13						
12	3e-07	3e-14						
13	8e-08							
14	2e-08							
15	5e-09							
16	1e-09							
17	3e-10							
18	8e-11							
19	2e-11							
20	5e-12							
21	1e-12							

Vergleicht man die Fehler von Trapez und Simpson, so sieht man leicht, dass der Fehler bei Simpson immer ungefähr das Quadrat des Trapez-Fehlers ist. Man sieht hier sofort, dass Simpson ein Verfahren der zweifachen Fehlerordnung der Trapezregel sein muß: Trapez hat Ordnung 1, Simpson Ordnung 2 und Romberg Ordnung 4. Letzteres sieht man in dieser Tabelle auch sehr schön: Betrachtet man z.Bsp. $k = 6$, dann ist der Trapez-Fehler 10^{-3} , und bei Romberg haben wir $10^{-13} < 10^{-12} = (10^{-3})^4$.

Weiterhin ist die enorme Auswirkung der Fehlerterm - Elimination durch die Extrapolation bei Romberg zu sehen (Werte wieder für $k = 6$): Der initiale Wert der Trapezregel, der mit $2^k = 64$ Auswertungen des Integranden (teuer!) plus $2^k = 64$ arithmetischen Operationen berechnet wurde, weist einen Fehler von 10^{-3} für diesen Integranden auf. Romberg spendiert nun gerade einmal $2k + 1 = 13$ zusätzliche arithmetische Operationen bei der Extrapolation und reduziert damit den Fehler um vier Potenzen auf 10^{-13} !! Die Trapezregel müsste dafür mehr als $2^{21} = 2.097.152$ Funktionsauswertungen und nochmal so viele arithmetische Operationen aufwenden (auch mit dem Trick der Zusatzaufgabe).

[Aber das haben Sie sicher an den Laufzeiten Ihrer Programme gemerkt ...]

Bleibt noch die Beobachtung bei Romberg, das für den ersten Integranden der beste Näherungswert nicht auf der Diagonale R_k^k steht, sondern bei R_k^{k-1} ! Also führt der jeweils k -te Extrapolationsschritt nicht mehr zu einer Reduktion des Fehlers, sondern vergrößert ihn wieder. Diese Beobachtung kann nicht verallgemeinert werden (siehe Integrand 2) und kann nur bei einfachen Integralen gemacht werden, deren exakte Lösung einfach zu bestimmen ist.

Fehler für $\int_0^1 \sqrt{x^3} dx$											
k	Trapez	Simpson	Romberg								
1	3e-02	2e-03	2e-03								
2	7e-03	4e-04	4e-04	3e-04							
3	2e-03	8e-05	8e-05	5e-05	5e-05						
4	5e-04	1e-05	1e-05	9e-06	9e-06	9e-06					
5	1e-04	2e-06	2e-06	2e-06	2e-06	2e-06	2e-06				
6	3e-05	4e-07	4e-07	3e-07	3e-07	3e-07	3e-07	3e-07			
7	7e-06	8e-08	8e-08	5e-08	5e-08	5e-08	5e-08	5e-08	5e-08		
8	2e-06	1e-08	1e-08	9e-09	9e-09	8e-09	8e-09	8e-09	8e-09	8e-09	
9	5e-07	2e-09	2e-09	2e-09	2e-09	1e-09	1e-09	1e-09	1e-09	1e-09	1e-09
10	1e-07	4e-10									
11	3e-08	7e-11									
12	7e-09	1e-11									
13	2e-09	2e-12									
14	5e-10	4e-13									
15	1e-10	7e-14									
16	3e-11										
17	7e-12										
18	2e-12										
19	4e-13										
20	9e-14										

Beim zweiten Integranden verhält sich Trapez gut, der Fehler wird sogar schneller kleiner als beim ersten Integranden. Simpson ist etwas langsamer als im ersten Fall, aber auch ok. Bei Romberg liefert jeweils nur der erste (!) Extrapolationsschritt (der lieferte aber gerade den Wert der Simpson-Regel) eine Reduktion des Fehlers, danach kann der Fehler für diese k nicht weiter reduziert werden. Der von Romberg berechnete Wert des Integrals hat einen deutlich größeren Fehler als der Wert von Trapez oder Simpson (Faktor $10^5 = 100.000$!!). D.h. der Wert von Trapez und Simpson ist 100.000 mal kleiner als der vom Romberg-Verfahren, Simpson und Trapez sind hier deutlich besser als Romberg.

Das ist ein enormen Unterschied und zeigt, daß die Voraussetzungen numerischer Algorithmen nicht einfach ignoriert werden dürfen. Denn nur wenn die Voraussetzungen eines Verfahrens vom zu lösenden Problem erfüllt werden, kann das Verfahren auch verlässliche Ergebnisse liefern.

Und die hier verletzte Voraussetzung findet sich auf Folie 102, *Approximationsfehler Romberg-Verfahren*: Dort wird der Fehler für das k -te Diagonalelement des Romberg-Schemas mit der Ordnung $2k + 2$ abgeschätzt, wenn die zu integrierende Funktion f auch $2k + 2$ -mal stetig differenzierbar im gesamten Integrationsintervall $[a, b] = [0, 1]$ ist.

Betrachten wir also die Funktion $f(x) = \sqrt{x^3}$ und ihre Ableitungen:

$$f(x) = x^{3/2} \quad f'(x) = \frac{3}{2} x^{1/2} \quad f''(x) = \frac{3}{4} x^{-1/2} = \frac{3}{4\sqrt{x}} \quad f'''(x) = \frac{-3}{8\sqrt{x^3}} \quad \dots$$

Da die Null zum Integrationsintervall gehört, ist leider ab $k = 1$ die Ableitung nicht stetig, damit ist die Voraussetzung nicht erfüllt und somit die Fehlerabschätzung nicht mehr gültig. Und genau das sieht man ja in der obigen Tabelle eindrucksvoll.

Übrigens hat die *Verlangsamung* von Simpson natürlich die gleiche Ursache: der Integrand ist nicht glatt genug.

Erstelle für jeden Integranden ein Diagramm, in dem der negative Logarithmus des Fehlers von Trapezregel, Simpson-Regel und Romberg - Verfahren zum exakten Wert des Integrals über dem jeweiligen k der zugehörigen Schrittweite h_k dargestellt wird. Wenn $I = \int_a^b f(x)dx$ den exakten Wert des jeweiligen Integrals bezeichnet, so sind also (soweit vorhanden) die Werte

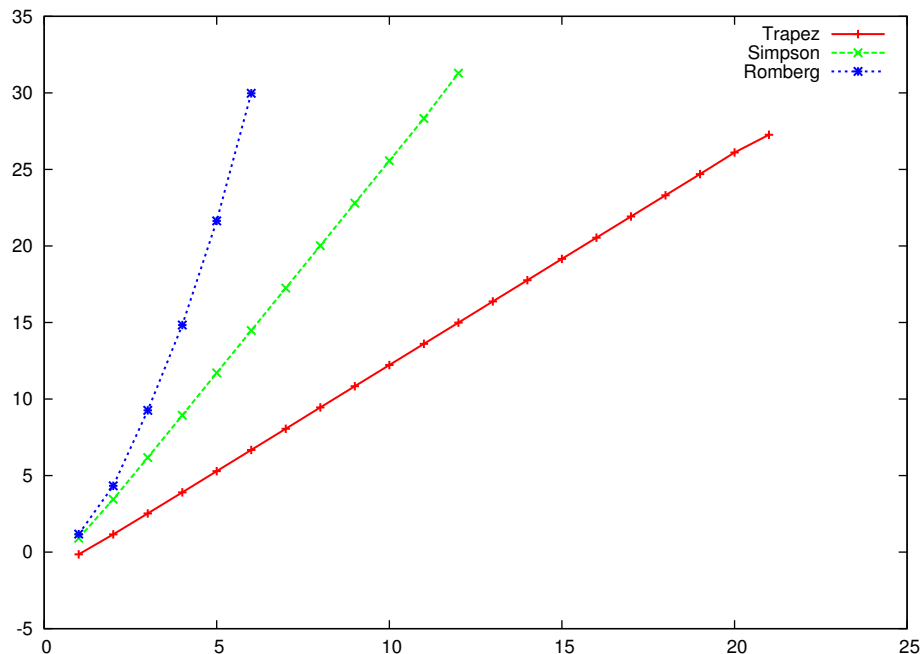
$$-\log |T_{2^k} - I|, \quad -\log |S_{2^k} - I|, \quad -\log |R_k^k - I|$$

für $k = 1, 2, \dots$ zu plotten. Bewerte das Verhalten der Grafen. Welche Bedeutung könnten diese Werte haben?

$\int_0^{\pi/2} e^{2x} \cos(x) dx$			
k	Trapez	Simpson	Romberg
1	-0.151769	0.905661	1.173955
2	1.156068	3.452550	4.330092
3	2.522694	6.173568	9.256434
4	3.904070	8.933753	14.830319
5	5.289135	11.703274	21.640378
6	6.675122	14.475099	29.975228
7	8.061340	17.247496	
8	9.447615	20.020037	
9	10.833904	22.792615	
10	12.220197	25.564789	
11	13.606492	28.331210	
12	14.992786	31.290063	
13	16.379080		
14	17.765375		
15	19.151676		
16	20.537956		
17	21.924285		
18	23.309827		

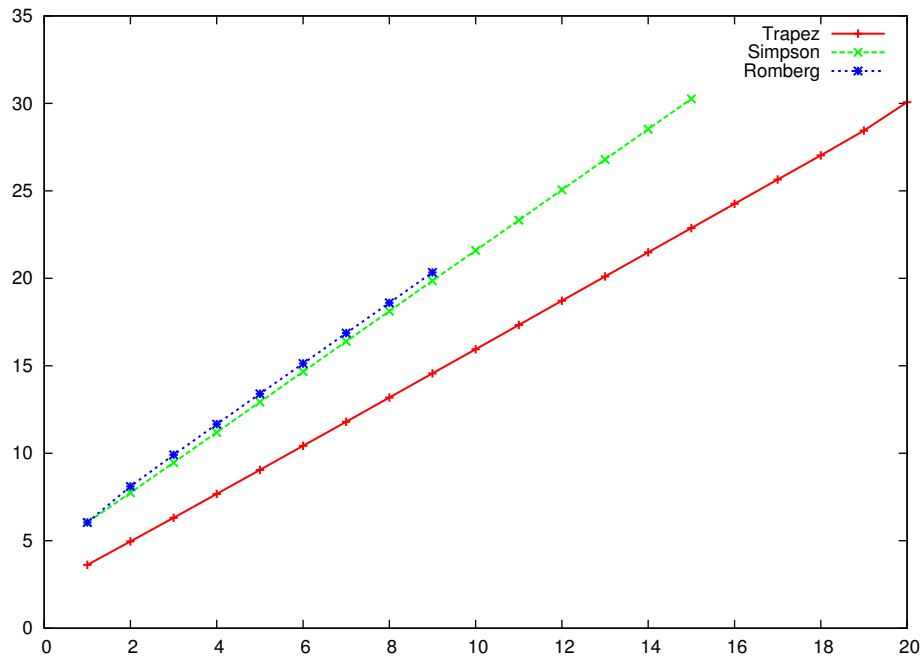
$\int_0^1 \sqrt{x^3} dx$			
k	Trapez	Simpson	Romberg
1	3.620223	6.045318	6.045318
2	4.959261	7.747279	8.102498
3	6.313068	9.468471	9.910516
4	7.676917	11.197132	11.661771
5	9.047616	12.928506	13.399201
6	10.423022	14.660846	15.133207
7	11.801687	16.393527	16.866359
8	13.182621	18.126329	18.599298
9	14.565142	19.859173	20.332184
10	15.948777	21.592033	
11	17.333195	23.324905	
12	18.718164	25.057793	
13	20.103523	26.791068	
14	21.489162	28.525318	
15	22.874991	30.255989	
16	24.260796		
17	25.647705		
18	27.030362		
19	28.447136		
20	30.066034		

1. Integrand: $\int_0^{\pi/2} e^{2x} \cos(x) dx$



Alle Werte bilden näherungsweise eine Gerade mit dem Anstieg der Verfahrensordnung, die die Theorie vorhersagt (Trapez 1, Simpson 2, Romberg 4).

2. Integrand: $\int_0^1 \sqrt{x^3} dx$



Man erkennt sehr schön, wie der nicht hinreichend glatte zweite Integrand die Reduktion des Fehlers mit wachsendem k behindert, also Simpson und erst recht Romberg gar nicht ihre volle Ordnung erreichen.

Bedeutung der Werte: Da der 2-er Logarithmus verwendet wurde, gibt der negative Logarithmus des Fehlers die ungefähre Anzahl der korrekt berechneten Bits an. (Entsprechend würde der 10-Logarithmus die Anzahl der korrekten Dezimalstellen liefern.)

Zusatzaufgabe: Bei der Berechnung der Trapezregel mit halbiertem Schrittweite $h_k = h_{k-1}/2$ kann der Berechnungsaufwand gesenkt werden, indem bei Schrittweite h_{k-1} berechnete Werte wieder benutzt werden. Gib eine Formel zur Berechnung von T_{2^k} unter Verwendung von $T_{2^{k-1}}$ an.

$$T_{2^k} = T_{2^{k-1}}/2 + \sum_{i=1}^{2^{k-1}} f_{2i-1}$$

oder in der Notation des Skripts

$$T_n = T_{\lfloor n/2 \rfloor} / 2 + \sum_{i=1}^{n/2} f_{2i-1}$$

Quelltexte:

1. Integrand (Datei f1.inc) Enthält Funktionen $f()$, $init_ab$ und $exact()$

```

1 double f( double x ) {
2     double pi = 4*atan(1);
3     return exp(2*x)*cos(x);
4 }
5
6 double init_ab( double *a, double *b ) {
7     double pi = 4*atan(1);
8     *a = 0; *b = pi/2;
9 }
10
11 double exact( double a, double b ) {
12     double pi = 4*atan(1);

```

```

13     double aa, bb;
14     init_ab(&aa,&bb);
15     if ( (a!=aa) || (b!=bb) )
16         fprintf(stderr, "Exact value available for a=%lf and b=%lf only !!!\n"
17                 "But you asked for a=%lf and b=%lf\n", aa, bb, a, b);
18     return (exp(pi)-2) / 5;
19 }

```

2. Integrand (Datei f2.inc) Enthält Funktionen f(), init_ab und exact()

```

1 double f( double x ) {
2     return sqrt(pow(x,3.0));
3 }
4
5 double init_ab( double *a, double *b ) {
6     *a = 0; *b = 1;
7 }
8
9 double exact( double a, double b ) {
10    double aa, bb;
11    init_ab(&aa,&bb);
12    if ( (a!=aa) || (b!=bb) )
13        fprintf(stderr, "Exact value available for a=%lf and b=%lf only !!!\n"
14                "But you asked for a=%lf and b=%lf\n", aa, bb, a, b);
15    return 2*(sqrt(pow(b,5.0)) - sqrt(pow(a,5.0)))/5;
16 }

```

Definition von δ (Datei delta.inc)

```

1     double delta = 1e-12;

```

Trapez-Regel

```

1 #include <math.h>
2 #include <stdio.h>
3
4 #include "output.inc"
5 #include "f.inc"
6 #include "delta.inc"
7
8 int main ( void ) {
9
10    double a, b, h, T, T0, delta, exactval;
11    int kk, n, w, i, k;
12
13    init_ab(&a,&b);
14    printf("# Trapez delta=%e", delta);
15    exactval = exact(a,b);
16    printf("# Exakt:%e\n", exactval);
17
18    n = 2;
19    h = (b - a) / n;
20    T = 0; T0 = f(a)+f(b);
21    // Trapez
22    for ( k=1; fabs(T-T0) > delta * fabs(T0); k++ ) {
23        T0 = T;
24        T = (f(a)+f(b))/2.0;
25        for ( i = 1; i < n ; i++ )
26            T += f(a+i*h);
27        T *= h;
28        printf("%2d", k); output(T, exactval);
29
30        // prepare next k
31        n *= 2;

```

```

32     h = ( b - a ) / n;
33     printf("\n");
34 }
35 }

```

Simpson-Regel

```

1  #include <math.h>
2  #include <stdio.h>
3
4  #include "output.inc"
5  #include "f.inc"
6  #include "delta.inc"
7
8  int main (void ) {
9
10     double a, b, h, T, T0, delta, exactval;
11     int kk, n, w, i, k;
12
13     init_ab(&a,&b);
14     printf("# Simpson delta=%e", delta);
15     exactval = exact(a,b);
16     printf("# Exakt: %e\n", exactval);
17
18     n = 2;
19     h = (b - a) / 2;
20     T = 0; T0 = f(a) + f(b);
21     // Simpson
22     for ( k=1; fabs(T-T0) > delta * fabs(T0); k++ ) {
23         T0 = T;
24         T = 0;
25         for ( i = 2; i < n ; i+=2 )
26             T += f(a+i*h) + 2 * f(a+(i+1)*h);
27         T = f(a) + 4*f(a+h) + 2*T + f(b);
28         T *= h / 3;
29         printf("%2d", k); output(T, exactval);
30
31         // prepare next k
32         n *= 2;
33         h = ( b - a ) / n;
34         printf("\n");
35     }
36 }

```

Romberg

```

1  #include <math.h>
2  #include <stdio.h>
3
4  #include "output.inc"
5  #include "f.inc"
6  #include "delta.inc"
7
8
9  int main (void ) {
10     double R[MAXK], s1, s2;
11     double a, b, h, T, T0, delta, exactval;
12     int kk, n, w, i, k;
13
14     init_ab(&a,&b);
15     printf("# Romberg delta=%e", delta);
16     exactval = exact(a,b);
17     printf("# Exakt: %e\n", exactval);

```

```

18     n = 2;
19     h = (b - a) / n;
20     T = 0; T0 = 5*delta;
21
22     R[0] = (f(a)+f(b))/2.0;
23     T = R[0]; T0 = 5 * T;
24
25     for ( k=1; fabs(T-T0) > delta * R[0]; k++ ) {
26         R[k] = 0.0;
27         // TRAPEZ
28         s1 = (f(a)+f(b))/2.0;
29         for ( i = 1; i < n ; i++ )
30             s1 += f(a+i*h);
31         s1 *= h;
32         T0 = T; T = s1;
33         printf("%2d_", k);
34         output(s1, exactval);
35         // ROMBERG
36         w = 4;
37         for ( i = 1; i <= k; i++ ) {
38             s2 = ( w*s1 - R[i-1] ) / ( w-1 );
39             R[i-1] = s1;
40             s1 = s2;
41             w *= 4;
42             T0 = T; T = s1;
43             output(s1, exactval);
44         } // ROMBERG loop
45         R[k] = s1;
46         // prepare next k
47         n *= 2;
48         h = ( b - a ) / n;
49         printf("\n");
50     }
51 }

```