Numerische Methoden der Physik

5 Gewöhnliche Differentialgleichungen, Randwertprobleme

Marc Wagner

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

SS 2014

5.2.1 Beispiel: QM, 1D unendlicher Potentialtopf

- QM, 1D unendlicher Potentialtopf:
  - Schrödinger-Gleichung ist eine Differentialgleichung 2. Ordnung mit unbekanntem Energieeigenwert $\hat{E}$:

$$-\psi''(\hat{x}) \quad = \quad \hat{E}\psi(\hat{x}).$$

  - Umschreiben in ein System dreier Differentialgleichungen 1. Ordnung:

$$\mathbf{y}'(\hat{x}) \quad = \quad \mathbf{f}(\mathbf{y}(\hat{x}), \hat{x}) \quad , \quad \mathbf{y}(\hat{x}) \quad = \quad (\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) \quad = \quad (\phi(\hat{x}), -\hat{E}($$

- C-Programm:

```
1. // **********
2.
3.
4.
5. // shooting.C
6.
7. // Berechnet die Energieeigenwerte des unendlichen Potentialtopfes,
8. // -\psi'' = E \psi   mit   \psi(x=0) = \psi(x=1) = 0 .
9.
```

```
10.
11.
12. // **********
13.
14.
15.
16. #include <math.h>
17. #include <stdio.h>
18. #include <stdlib.h>
19.
20.
21.
22. // **********
23.
24.
25.
26. // ********************
27. // Physikalische Parameter, etc.
28. // ********************
29.
30.
31. // Schroedinger-Gleichung:
32. //   y = (\psi , \phi , E)
33. //   f = (\phi , -E \psi , 0)
34.
35.
36. const int N = 3;  // Anzahl der Komponenten von y bzw. f.
```

```
37.
38.
39. double y_0[N] = { 0.0 , 1.0 , 9.0 };  // Anfangsbedingungen y(t=0).
40.
41.
42. // Berechnet f(y(t),t) * tau.
43.
44. void f_times_tau(double *y_t, double t, double *f_times_tau_, double tau)
45. {
46.   if(N != 3)
47.     {
48.       fprintf(stderr, "Error: N != 3!\n");
49.       exit(EXIT_FAILURE);
50.     }
51.
52.   f_times_tau_[0] = y_t[1] * tau;
53.
54.   f_times_tau_[1] = -y_t[2] * y_t[0] * tau;
55.
56.   f_times_tau_[2] = 0.0;
57. }
58.
59.
60.
61. // **********
62.
63.
```

```
64.
65. // ********************
66. // RK-Parameter.
67. // ********************
68.
69.
70. // #define __EULER__
71. // #define __RK_2ND__
72. // #define __RK_3RD__
73. #define __RK_4TH__
74.
75. #ifdef __EULER__
76. const int order = 1;
77. #endif
78.
79. #ifdef __RK_2ND__
80. const int order = 2;
81. #endif
82.
83. #ifdef __RK_3RD__
84. const int order = 3;
85. #endif
86.
87. #ifdef __RK_4TH__
88. const int order = 4;
89. #endif
90.
```

```
 91. // Anzahl der RK-Schritte.
 92. const int num_steps = 1000;
 93.
 94. // Integration von t_0 bis t_1.
 95. const double t_0 = 0.0;
 96. const double t_1 = 1.0;
 97.
 98. double tau = (t_1 - t_0) / (double)num_steps;  // Schrittweite.
 99.
100. double h = 0.000001;  // Diskretisierungsschritt fuer numerische Ableitung.
101.
102. double dE_min = 0.0000001;  // Abbruchkriterium fuer Newton-Raphson.
103.
104.
105. // **********
106.
107.
108.
109. #ifdef __EULER__
110.
111. // Euler-Schritt um Schrittweite tau.
112.
113. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
114. {
115.   int i1;
116.
117.   // *****
```

```
118.
119.   // Berechne k1 = f(y(t),t) * tau.
120.
121.   double k1[N];
122.   f_times_tau(y_t, t, k1, tau);
123.
124.   // *****
125.
126.   for(i1 = 0; i1 < N; i1++)
127.     y_t_plus_tau[i1] = y_t[i1] + k1[i1];
128. }
129.
130. #endif
131.
132.
133.
134. #ifdef __RK_2ND__
135.
136. // RK-Schritt (2nd order) um Schrittweite tau.
137.
138. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
139. {
140.   int i1;
141.
142.   // *****
143.
144.   // Berechne k1 = f(y(t),t) * tau.
```

```
145.
146.   double k1[N];
147.   f_times_tau(y_t, t, k1, tau);
148.
149.   // *****
150.
151.   // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.
152.
153.   double y_[N];
154.
155.   for(i1 = 0; i1 < N; i1++)
156.     y_[i1] = y_t[i1] + 0.5*k1[i1];
157.
158.   double k2[N];
159.   f_times_tau(y_, t + 0.5*tau, k2, tau);
160.
161.   // *****
162.
163.   for(i1 = 0; i1 < N; i1++)
164.     y_t_plus_tau[i1] = y_t[i1] + k2[i1];
165. }
166.
167. #endif
168.
169.
170.
171. #ifdef __RK_3RD__
```

```
172.
173. // RK-Schritt (3rd order) um Schrittweite tau.
174.
175. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
176. {
177.   int i1;
178.
179.   // *****
180.
181.   // Berechne k1 = f(y(t),t) * tau.
182.
183.   double k1[N];
184.   f_times_tau(y_t, t, k1, tau);
185.
186.   // *****
187.
188.   // Berechne k2 = f(y(t)+k1 , t+tau) * tau.
189.
190.   double y_[N];
191.
192.   for(i1 = 0; i1 < N; i1++)
193.     y_[i1] = y_t[i1] + k1[i1];
194.
195.   double k2[N];
196.   f_times_tau(y_, t + tau, k2, tau);
197.
198.   // *****
```

```
199.
200.   // Berechne k3 = f(y(t)+(1/4)*(k1+k2) , t+(1/2)*tau) * tau.
201.
202.   for(i1 = 0; i1 < N; i1++)
203.     y_[i1] = y_t[i1] + 0.25 * (k1[i1] + k2[i1]);
204.
205.   double k3[N];
206.   f_times_tau(y_, t + 0.5*tau, k3, tau);
207.
208.   // *****
209.
210.   for(i1 = 0; i1 < N; i1++)
211.     y_t_plus_tau[i1] = y_t[i1] + (k1[i1] + k2[i1] + 4.0*k3[i1]) / 6.0;
212. }
213.
214. #endif
215.
216.
217.
218. #ifdef __RK_4TH__
219.
220. // RK-Schritt (4th order) um Schrittweite tau.
221.
222. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
223. {
224.   int i1;
225.
```

```
226.  // *****
227.
228.  // Berechne k1 = f(y(t),t) * tau.
229.
230.  double k1[N];
231.  f_times_tau(y_t, t, k1, tau);
232.
233.  // *****
234.
235.  // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.
236.
237.  double y_[N];
238.
239.  for(i1 = 0; i1 < N; i1++)
240.    y_[i1] = y_t[i1] + 0.5*k1[i1];
241.
242.  double k2[N];
243.  f_times_tau(y_, t + 0.5*tau, k2, tau);
244.
245.  // *****
246.
247.  // Berechne k3 = f(y(t)+(1/2)*k2 , t+(1/2)*tau) * tau.
248.
249.  for(i1 = 0; i1 < N; i1++)
250.    y_[i1] = y_t[i1] + 0.5*k2[i1];
251.
252.  double k3[N];
```

```
253.   f_times_tau(y_, t + 0.5*tau, k3, tau);
254. 
255.   // *****
256. 
257.   // Berechne k4 = f(y(t)+k3 , t+tau) * tau.
258. 
259.   for(i1 = 0; i1 < N; i1++)
260.     y_[i1] = y_t[i1] + k3[i1];
261. 
262.   double k4[N];
263.   f_times_tau(y_, t + tau, k4, tau);
264. 
265.   // *****
266. 
267.   for(i1 = 0; i1 < N; i1++)
268.     {
269.       y_t_plus_tau[i1] =
270.         y_t[i1] + (k1[i1] + 2.0*k2[i1] + 2.0*k3[i1] + k4[i1]) / 6.0;
271.     }
272. }
273. 
274. #endif
275. 
276. 
277. 
278. // **********
279. 
```

```
280.
281.
282. // RK-Entwicklung der "Wellenfunktion".
283.
284. double t[num_steps+1];  // Diskretisierte "Zeitachse".
285. double y[num_steps+1][N];  // Diskretisierte "Bahnkurven".
286.
287. double RK(bool output = false)
288. {
289.   double d1;
290.   int i1, i2;
291.
292.   // *****
293.
294.   // Fuehre Euler/RK-Schritte aus.
295.
296.   for(i1 = 0; i1 < num_steps; i1++)
297.     {
298.       // y(t)   -->   y(t+\tau)
299.       RK_step(y[i1], t[i1], y[i1+1], tau);
300.
301.       t[i1+1] = t[i1] + tau;
302.     }
303.
304.   // *****
305.
306.   if(output == true)
```

```
307.    {
308.      // Ausgabe.
309.
310.      for(i1 = 0; i1 <= num_steps; i1++)
311.        {
312.          printf("%9.6lf %9.6lf %9.6lf %9.6lf\n",
313.                 t[i1], y[i1][0], y[i1][1], y[i1][2]);
314.        }
315.    }
316.
317.  // *****
318.
319.  return y[num_steps][0];
320. }
321.
322.
323.
324. // **********
325.
326.
327.
328. int main(int argc, char **argv)
329. {
330.   int i1;
331.
332.   // *****
333.
```

```
334.  // Initialisiere "Bahnkurven" mit Anfangsbedingungen.
335.
336.  t[0] = t_0;
337.
338.  for(i1 = 0; i1 < N; i1++)
339.    y[0][i1] = y_0[i1];
340.
341.  // **********
342.  // **********
343.  // **********
344.
345.  // Grobes Scannen der moeglichen Energieeigenwerte.
346.
347.  // **********
348.  // **********
349.  // **********
350.
351.  /*
352.  double E_min = 0.0;
353.  double E_max = 100.0;
354.
355.  double E_step = 5.0;
356.
357.  for(double E = E_min; E <= E_max; E += E_step)
358.    {
359.      // Modifiziere Anfangsbedingungen.
360.      y[0][N-1] = E;
```

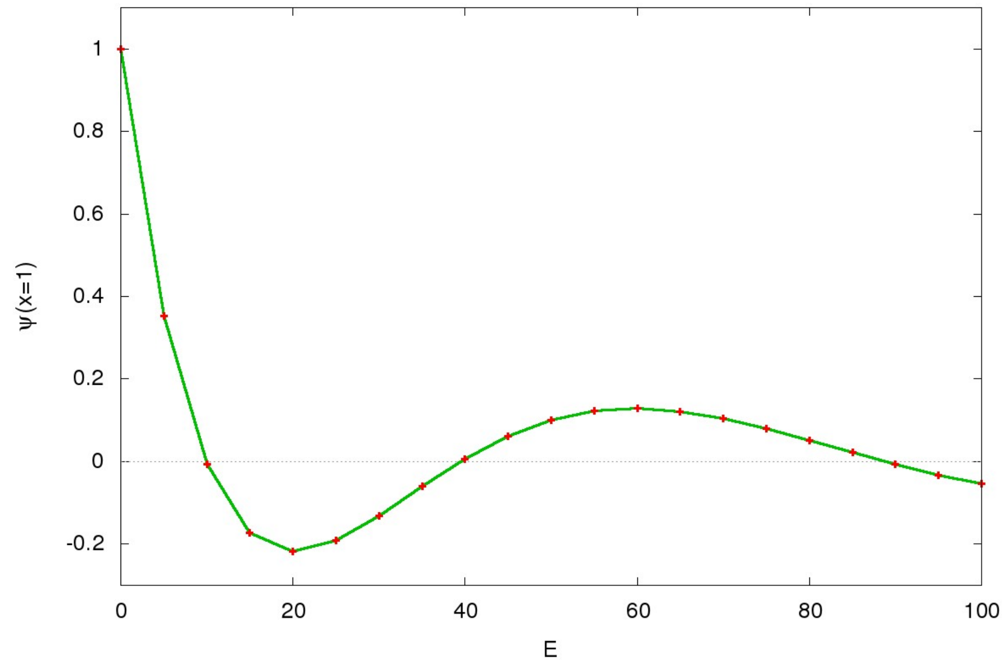
```
361.
362.       // Fuehre Euler/RK-Schritte aus.
363.       double psi_1 = RK(false);
364.
365.       // *****
366.
367.       printf("%+.5e %+.5e.\n", E, psi_1);
368.     }
369.   */
370.
371.   // **********
372.   // **********
373.   // **********
374.
375.   // **********
376.   // **********
377.   // **********
378.
379.   // Newton-Raphson zur praezisen Bestimmung einzelner Energieeigenwerte.
380.
381.   // **********
382.   // **********
383.   // **********
384.
385.   // /*
386.   // double E = 10.0;  // Initialer Schätzwert fuer Energieeigenwert.
387.   // double E = 40.0;  // Initialer Schätzwert fuer Energieeigenwert.
```

```
388.  double E = 90.0;  // Initialer Schätzwert fuer Energieeigenwert.
389.
390.  fprintf(stderr, "E_num = %+10.6lf .\n", E);
391.
392.  while(1)
393.    {
394.      // Modifiziere Anfangsbedingungen.
395.      y[0][N-1] = E;
396.
397.      // Fuehre Euler/RK-Schritte aus.
398.      double psi_1_E = RK(false);
399.
400.      // *****
401.
402.      // Modifiziere Anfangsbedingungen.
403.      y[0][N-1] = E-h;
404.
405.      // Fuehre Euler/RK-Schritte aus.
406.      double psi_1_E_mi_h = RK(false);
407.
408.      // Modifiziere Anfangsbedingungen.
409.      y[0][N-1] = E+h;
410.
411.      // Fuehre Euler/RK-Schritte aus.
412.      double psi_1_E_pl_h = RK(false);
413.
414.      double dpsi_1_E = (psi_1_E_pl_h - psi_1_E_mi_h) / (2.0 * h);
```

```
415.
416.        // *****
417.
418.        // Newton-Raphson-Schritt.
419.
420.        double dE = psi_1_E / dpsi_1_E;
421.
422.        if(fabs(dE) < dE_min)
423.          break;
424.
425.        E = E - dE;
426.
427.        // *****
428.
429.        // fprintf(stderr, "E_num = %+10.6lf  ,  E_ana = %+10.6lf  ,  \\psi(x=1) = %+.6lf .\n", E, M_PI*M_PI, psi_1_E);
430.        // fprintf(stderr, "E_num = %+10.6lf  ,  E_ana = %+10.6lf  ,  \\psi(x=1) = %+.6lf .\n", E, 4.0*M_PI*M_PI, psi_1_E);
431.        fprintf(stderr, "E_num = %+10.6lf  ,  E_ana = %+10.6lf  ,  \\psi(x=1) = %+.6lf .\n", E, 9.0*M_PI*M_PI, psi_1_E);
432.      }
433.
434.    // Ausgabe der Wellenfunktion.
435.    RK(true);
436.
437.    // */
438.
439.    // **********
440.    // **********
441.    // **********
```

```
442.
443.   return EXIT_SUCCESS;
444. }
445.
446.
447.
448. // **********
```

- Shooting mit Anfangsbedingungen
  $\psi(0) = 0$,
  $\phi(0) = 1$ (beliebig, Hauptsache $\neq 0$, betrifft nur Normierung),
  $\hat{E}(0) =?$ (wird mit Newton-Raphson so eingestellt, dass RB $\psi'(1) = 0$ erfüllt ist):
  - Grobes Scannen der der möglichen Energieeigenwerte: Im Bereich $0.0 \leq \hat{E} \leq 100.0$ liegen drei Energieeigenwerte, $\hat{E}_1 \approx 10.0$, $\hat{E}_1 \approx 40.0$, $\hat{E}_1 \approx 90.0$.



- Newton-Raphson-Iteration: Konvergiert bereits nach 3 Schritten auf über 7 Stellen genau.
    - Grundzustand:

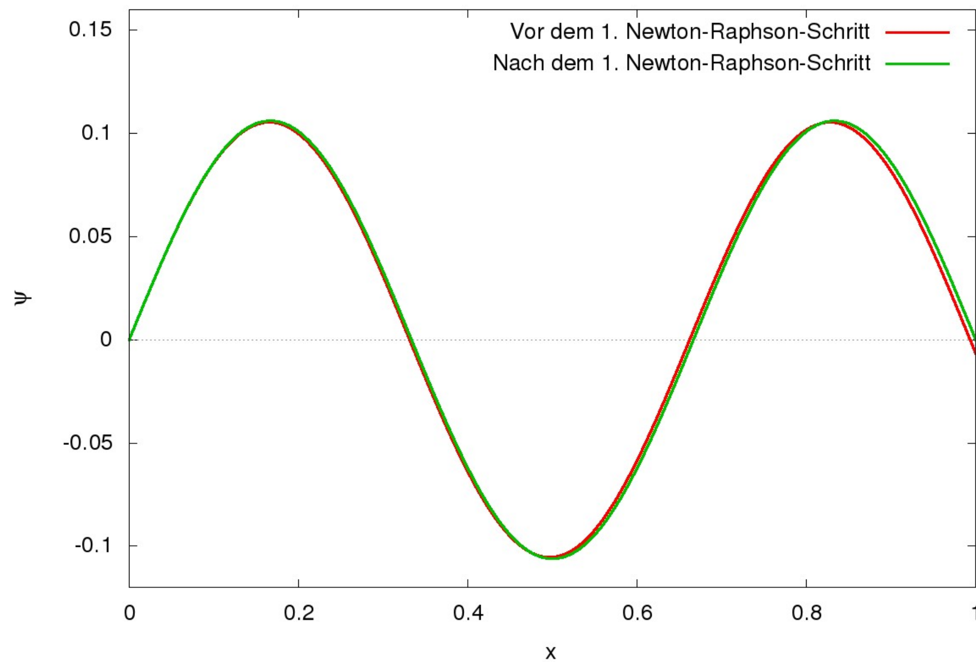
```
E_num = +10.000000 .
E_num =  +9.868296  ,  E_ana =  +9.869604  ,  \psi(x=1) = -0.006541 .
E_num =  +9.869604  ,  E_ana =  +9.869604  ,  \psi(x=1) = +0.000066 .
E_num =  +9.869604  ,  E_ana =  +9.869604  ,  \psi(x=1) = +0.000000 .
```

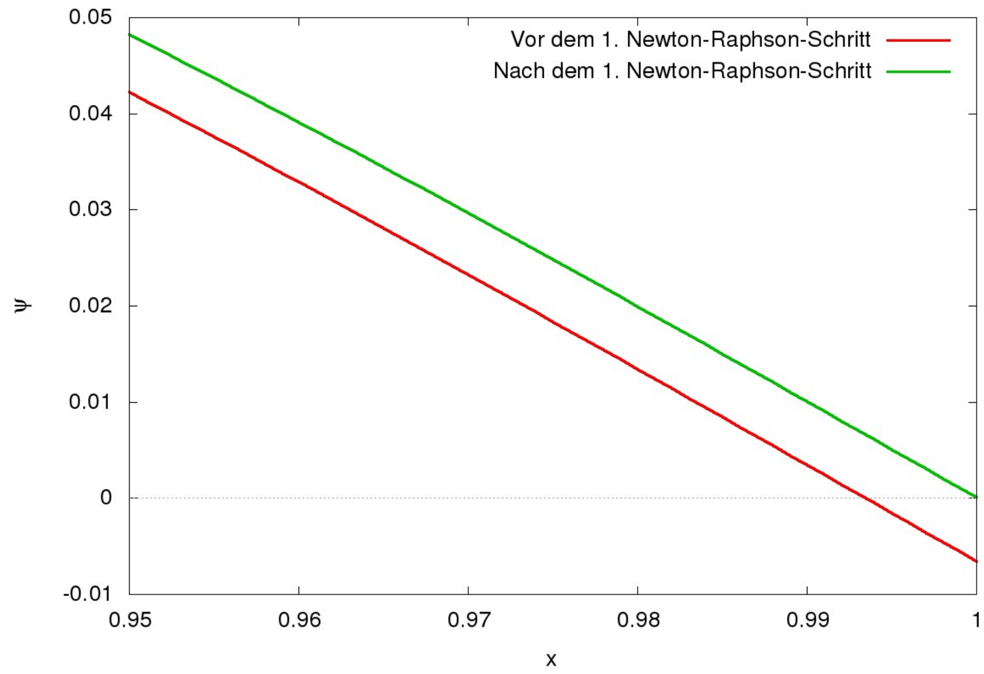
- 1. angeregter Zustand:

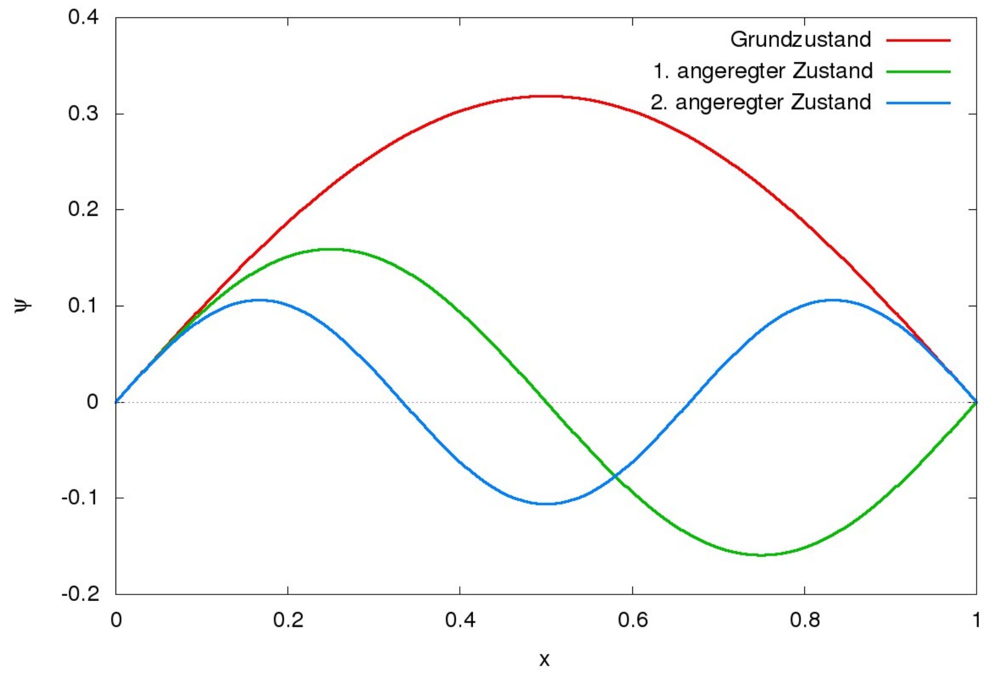
```
E_num = +40.000000 .
E_num = +39.472958   ,  E_ana = +39.478418   ,  \psi(x=1) = +0.006539 .
E_num = +39.478417   ,  E_ana = +39.478418   ,  \psi(x=1) = -0.000069 .
E_num = +39.478418   ,  E_ana = +39.478418   ,  \psi(x=1) = -0.000000 .
```

- 2. angeregter Zustand:

```
E_num = +90.000000 .
E_num = +88.813303   ,  E_ana = +88.826440   ,  \psi(x=1) = -0.006537 .
E_num = +88.826438   ,  E_ana = +88.826440   ,  \psi(x=1) = +0.000074 .
E_num = +88.826440   ,  E_ana = +88.826440   ,  \psi(x=1) = +0.000000 .
```




Potentialtopf: Iteratives Finden der Wellenfunktion des 2. angeregten Zustands
Vor dem 1. Newton-Raphson-Schritt
Nach dem 1. Newton-Raphson-Schritt
ψ
x
0.05
0.04
0.03
0.02
0.01
0
-0.01
0.95
0.96
0.97
0.98
0.99
1


Potentialtopf: (Nicht-normierte) Wellenfunktionen der niedrigsten Energieeigenzustaende
Grundzustand
1. angeregter Zustand
2. angeregter Zustand
ψ
x
0.4
0.3
0.2
0.1
0
-0.1
-0.2
0
0.2
0.4
0.6
0.8
1

5.2.2 Beispiel: QM, 1D harmonischer Oszillator

- QM, 1D harmonischer Oszillator:
  - Schrödinger-Gleichung ist eine Differentialgleichung 2. Ordnung mit unbekanntem Energieeigenwert $\hat{E}$:

$$-\psi''(\hat{x}) + \hat{x}^2 \psi(\hat{x}) = \hat{E}\psi(\hat{x}).$$

  - Umschreiben in ein System dreier Differentialgleichungen 1. Ordnung:

$$\mathbf{y}'(\hat{x}) = \mathbf{f}(\mathbf{y}(\hat{x}), \hat{x}) \quad , \quad \mathbf{y}(\hat{x}) = (\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) = (\phi(\hat{x}), \Big(\hat{x}^2$$

- **C**-Programm:

```
1. // **********
2.
3.
4.
5. // shooting.C
6.
7. // Berechnet die Energieeigenwerte des harmonischen Oszillators,
8. // -\psi'' = (E -x^2) \psi   mit   \psi(x=-\infty) = \psi(x=+\infty) = 0 .
```

```
9.
10.
11.
12. // **********
13.
14.
15.
16. #include <math.h>
17. #include <stdio.h>
18. #include <stdlib.h>
19.
20.
21.
22. // **********
23.
24.
25.
26. // ********************
27. // Physikalische Parameter, etc.
28. // ********************
29.
30.
31. // Schroedinger-Gleichung:
32. //   y = (\psi , \phi , E)
33. //   f = (\phi , (x^2-E) \psi , 0)
34.
35.
```

```
36. const int N = 3;  // Anzahl der Komponenten von y bzw. f.
37.
38.
39. // *****
40.
41.
42. // Plot 1
43.
44. // Integration von t_0 bis t_1.
45. // const double t_0 = 0.0;
46. // const double t_1 = 10.0;
47.
48. // Anfangsbedingungen y(t_0).
49. // double y_0[N] = { 1.0 , 0.0 , 1.0 - 0.000001};
50. // double y_0[N] = { 1.0 , 0.0 , 1.0 + 0.000001};
51.
52. // *****
53.
54.
55. // Plot 2
56.
57. // Integration von t_0 bis t_1.
58. const double t_0 = 10.0;
59. const double t_1 = 0.0;
60.
61. // Anfangsbedingungen y(t_0).
62. // double y_0[N] = { 1.0 , 0.0 , 5.0 };
```

```
63. double y_0[N] = { 0.0 , 1.0 , 5.0 };
64.
65.
66. // *****
67.
68.
69. // Berechnet f(y(t),t) * tau.
70.
71. void f_times_tau(double *y_t, double t, double *f_times_tau_, double tau)
72. {
73.   if(N != 3)
74.     {
75.       fprintf(stderr, "Error: N != 3!\n");
76.       exit(EXIT_FAILURE);
77.     }
78.
79.   f_times_tau_[0] = y_t[1] * tau;
80.
81.   f_times_tau_[1] = (t*t - y_t[2]) * y_t[0] * tau;
82.
83.   f_times_tau_[2] = 0.0;
84. }
85.
86.
87.
88. // **********
89.
```

```
90.
91.
92. // ********************
93. // RK-Parameter.
94. // ********************
95.
96.
97. // #define __EULER__
98. // #define __RK_2ND__
99. // #define __RK_3RD__
100. #define __RK_4TH__
101.
102. #ifdef __EULER__
103. const int order = 1;
104. #endif
105.
106. #ifdef __RK_2ND__
107. const int order = 2;
108. #endif
109.
110. #ifdef __RK_3RD__
111. const int order = 3;
112. #endif
113.
114. #ifdef __RK_4TH__
115. const int order = 4;
116. #endif
```

```
117.
118. // Anzahl der RK-Schritte.
119. const int num_steps = 1000;
120.
121. double tau = (t_1 - t_0) / (double)num_steps;  // Schrittweite.
122.
123. double h = 0.000001;  // Diskretisierungsschritt fuer numerische Ableitung.
124.
125. double dE_min = 0.0000001;  // Abbruchkriterium fuer Newton-Raphson.
126.
127.
128. // **********
129.
130.
131.
132. #ifdef __EULER__
133.
134. // Euler-Schritt um Schrittweite tau.
135.
136. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
137. {
138.   int i1;
139.
140.   // *****
141.
142.   // Berechne k1 = f(y(t),t) * tau.
143.
```

```
144.   double k1[N];
145.   f_times_tau(y_t, t, k1, tau);
146.
147.   // *****
148.
149.   for(i1 = 0; i1 < N; i1++)
150.     y_t_plus_tau[i1] = y_t[i1] + k1[i1];
151. }
152.
153. #endif
154.
155.
156.
157. #ifdef __RK_2ND__
158.
159. // RK-Schritt (2nd order) um Schrittweite tau.
160.
161. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
162. {
163.   int i1;
164.
165.   // *****
166.
167.   // Berechne k1 = f(y(t),t) * tau.
168.
169.   double k1[N];
170.   f_times_tau(y_t, t, k1, tau);
```

```
171.
172.   // *****
173.
174.   // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.
175.
176.   double y_[N];
177.
178.   for(i1 = 0; i1 < N; i1++)
179.     y_[i1] = y_t[i1] + 0.5*k1[i1];
180.
181.   double k2[N];
182.   f_times_tau(y_, t + 0.5*tau, k2, tau);
183.
184.   // *****
185.
186.   for(i1 = 0; i1 < N; i1++)
187.     y_t_plus_tau[i1] = y_t[i1] + k2[i1];
188. }
189.
190. #endif
191.
192.
193.
194. #ifdef __RK_3RD__
195.
196. // RK-Schritt (3rd order) um Schrittweite tau.
197.
```

```
198. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
199. {
200.   int i1;
201.
202.   // *****
203.
204.   // Berechne k1 = f(y(t),t) * tau.
205.
206.   double k1[N];
207.   f_times_tau(y_t, t, k1, tau);
208.
209.   // *****
210.
211.   // Berechne k2 = f(y(t)+k1 , t+tau) * tau.
212.
213.   double y_[N];
214.
215.   for(i1 = 0; i1 < N; i1++)
216.     y_[i1] = y_t[i1] + k1[i1];
217.
218.   double k2[N];
219.   f_times_tau(y_, t + tau, k2, tau);
220.
221.   // *****
222.
223.   // Berechne k3 = f(y(t)+(1/4)*(k1+k2) , t+(1/2)*tau) * tau.
224.
```

```
225.   for(i1 = 0; i1 < N; i1++)
226.     y_[i1] = y_t[i1] + 0.25 * (k1[i1] + k2[i1]);
227.
228.   double k3[N];
229.   f_times_tau(y_, t + 0.5*tau, k3, tau);
230.
231.   // *****
232.
233.   for(i1 = 0; i1 < N; i1++)
234.     y_t_plus_tau[i1] = y_t[i1] + (k1[i1] + k2[i1] + 4.0*k3[i1]) / 6.0;
235. }
236.
237. #endif
238.
239.
240.
241. #ifdef __RK_4TH__
242.
243. // RK-Schritt (4th order) um Schrittweite tau.
244.
245. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
246. {
247.   int i1;
248.
249.   // *****
250.
251.   // Berechne k1 = f(y(t),t) * tau.
```

```
252.
253.   double k1[N];
254.   f_times_tau(y_t, t, k1, tau);
255.
256.   // *****
257.
258.   // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.
259.
260.   double y_[N];
261.
262.   for(i1 = 0; i1 < N; i1++)
263.     y_[i1] = y_t[i1] + 0.5*k1[i1];
264.
265.   double k2[N];
266.   f_times_tau(y_, t + 0.5*tau, k2, tau);
267.
268.   // *****
269.
270.   // Berechne k3 = f(y(t)+(1/2)*k2 , t+(1/2)*tau) * tau.
271.
272.   for(i1 = 0; i1 < N; i1++)
273.     y_[i1] = y_t[i1] + 0.5*k2[i1];
274.
275.   double k3[N];
276.   f_times_tau(y_, t + 0.5*tau, k3, tau);
277.
278.   // *****
```

```
279.
280.   // Berechne k4 = f(y(t)+k3 , t+tau) * tau.
281.
282.   for(i1 = 0; i1 < N; i1++)
283.     y_[i1] = y_t[i1] + k3[i1];
284.
285.   double k4[N];
286.   f_times_tau(y_, t + tau, k4, tau);
287.
288.   // *****
289.
290.   for(i1 = 0; i1 < N; i1++)
291.     {
292.       y_t_plus_tau[i1] =
293.         y_t[i1] + (k1[i1] + 2.0*k2[i1] + 2.0*k3[i1] + k4[i1]) / 6.0;
294.     }
295. }
296.
297. #endif
298.
299.
300.
301. // **********
302.
303.
304.
305. // RK-Entwicklung der "Wellenfunktion".
```

```
306.
307. double t[num_steps+1];  // Diskretisierte Zeitachse.
308. double y[num_steps+1][N];  // Diskretisierte "Bahnkurven".
309.
310. double RK(bool output = false)
311. {
312.   double d1;
313.   int i1, i2;
314.
315.   // *****
316.
317.   // Fuehre Euler/RK-Schritte aus.
318.
319.   for(i1 = 0; i1 < num_steps; i1++)
320.     {
321.       // y(t)   -->   y(t+\tau)
322.       RK_step(y[i1], t[i1], y[i1+1], tau);
323.
324.       t[i1+1] = t[i1] + tau;
325.     }
326.
327.   // *****
328.
329.   if(output == true)
330.     {
331.       // Ausgabe.
332.
```

```
333.       for(i1 = 0; i1 <= num_steps; i1++)
334.         {
335.           printf("%9.6lf %.5e %.5e %9.6lf\n",
336.                  t[i1], y[i1][0], y[i1][1], y[i1][2]);
337.         }
338.     }
339.
340.   // *****
341.
342.   // return y[num_steps][1];  // Fuer P = + (0., 2., 4., ... Zustand)
343.   return y[num_steps][0];  // Fuer P = + (1., 3., 5., ... Zustand)
344. }
345.
346.
347.
348. // **********
349.
350.
351.
352. int main(int argc, char **argv)
353. {
354.   int i1;
355.
356.   // *****
357.
358.   // Initialisiere "Bahnkurven" mit Anfangsbedingungen.
359.
```

```
360.   t[0] = t_0;
361.
362.   for(i1 = 0; i1 < N; i1++)
363.     y[0][i1] = y_0[i1];
364.
365.   // **********
366.   // **********
367.   // **********
368.
369.   // Einmalige RK-Entwicklung.
370.
371.   // **********
372.   // **********
373.   // **********
374.
375.   /*
376.   RK(true);
377.   */
378.
379.   // **********
380.   // **********
381.   // **********
382.
383.   // **********
384.   // **********
385.   // **********
386.
```

```
387.   // Grobes Scannen der moeglichen Energieeigenwerte.
388.
389.   // **********
390.   // **********
391.   // **********
392.
393.   /*
394.   double E_min = 0.0;
395.   double E_max = 12.0;
396.
397.   double E_step = 0.1;
398.
399.   for(double E = E_min; E <= E_max; E += E_step)
400.     {
401.       // Modifiziere Anfangsbedingungen.
402.       y[0][N-1] = E;
403.
404.       // Führe Euler/RK-Schritte aus.
405.       double psi_1 = RK(false);
406.
407.       // *****
408.
409.       printf("%+.5e %+.5e.\n", E, psi_1);
410.     }
411.   */
412.
413.   // **********
```

```
414.  // **********
415.  // **********
416.
417.  // **********
418.  // **********
419.  // **********
420.
421.  // Newton-Raphson zur praezisen Bestimmung einzelner Energieeigenwerte.
422.
423.  // **********
424.  // **********
425.  // **********
426.
427.  // /*
428.  // double E = 0.9;  // Initialer Schaetzwert fuer Energieeigenwert.
429.  // double E = 2.9;  // Initialer Schaetzwert fuer Energieeigenwert.
430.  // double E = 4.9;  // Initialer Schaetzwert fuer Energieeigenwert.
431.  double E = 6.9;  // Initialer Schätzwert fuer Energieeigenwert.
432.
433.  fprintf(stderr, "E_num = %+10.6lf .\n", E);
434.
435.  while(1)
436.    {
437.      // Modifiziere Anfangsbedingungen.
438.      y[0][N-1] = E;
439.
440.      // Führe Euler/RK-Schritte aus.
```

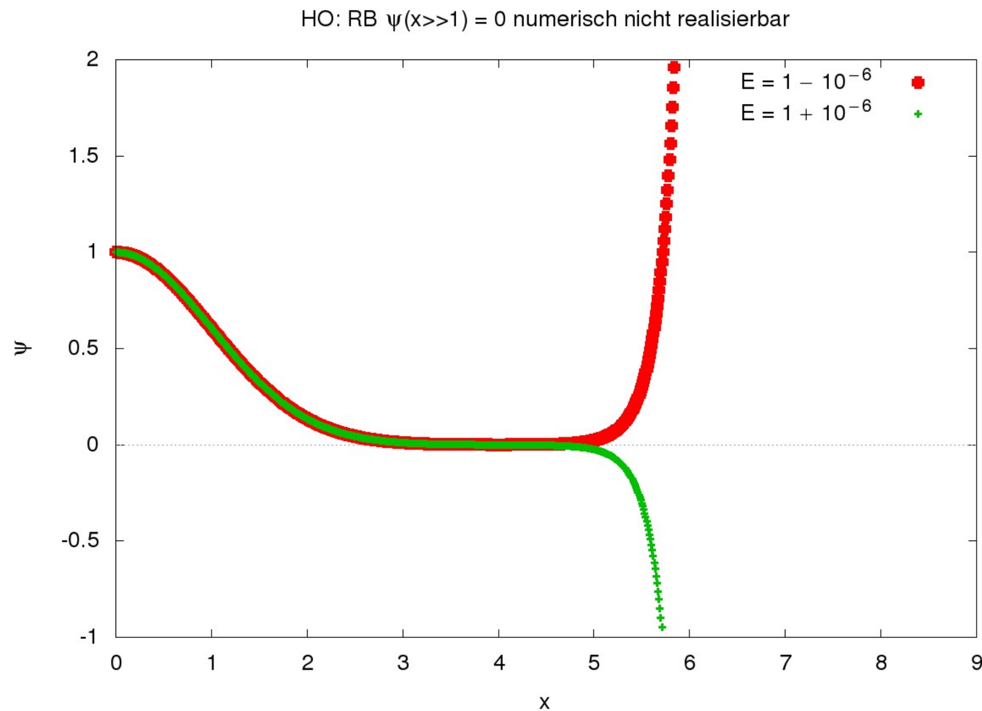
```
441.      double psi_1_E = RK(false);
442.
443.      // *****
444.
445.      // Modifiziere Anfangsbedingungen.
446.      y[0][N-1] = E-h;
447.
448.      // Fuehre Euler/RK-Schritte aus.
449.      double psi_1_E_mi_h = RK(false);
450.
451.      // Modifiziere Anfangsbedingungen.
452.      y[0][N-1] = E+h;
453.
454.      // Fuehre Euler/RK-Schritte aus.
455.      double psi_1_E_pl_h = RK(false);
456.
457.      double dpsi_1_E = (psi_1_E_pl_h - psi_1_E_mi_h) / (2.0 * h);
458.
459.      // *****
460.
461.      // Newton-Raphson-Schritt.
462.
463.      double dE = psi_1_E / dpsi_1_E;
464.
465.      if(fabs(dE) < dE_min)
466.        break;
467.
```

```
468.     E = E - dE;
469.
470.     // *****
471.
472.     fprintf(stderr, "E_num = %+10.6lf .\n", E);
473.   }
474.
475.   // Ausgabe der Wellenfunktion.
476.   RK(true);
477.
478.   // */
479.
480.   // **********
481.   // **********
482.   // **********
483.
484.   return EXIT_SUCCESS;
485. }
486.
487.
488.
489. // **********
```

- Shooting mit Anfangsbedingungen
  $\psi(0) = 1$ (beliebig, Hauptsache $\neq 0$, betrifft nur Normierung),
  $\phi(0) = 0$ (damit $P = +$),

$\hat{E}(0) = ?$ (wird mit Newton-Raphson so eingestellt, dass RB $\psi(L/a) = 0$ erfüllt ist):

- **RB $\psi(L/a) = 0$ numerisch nur schlecht realisierbar ... selbst eine winzige Beimischung der verbotenen exponentiell ansteigenden Lösung wird für große $\hat{x}$ dominieren.**

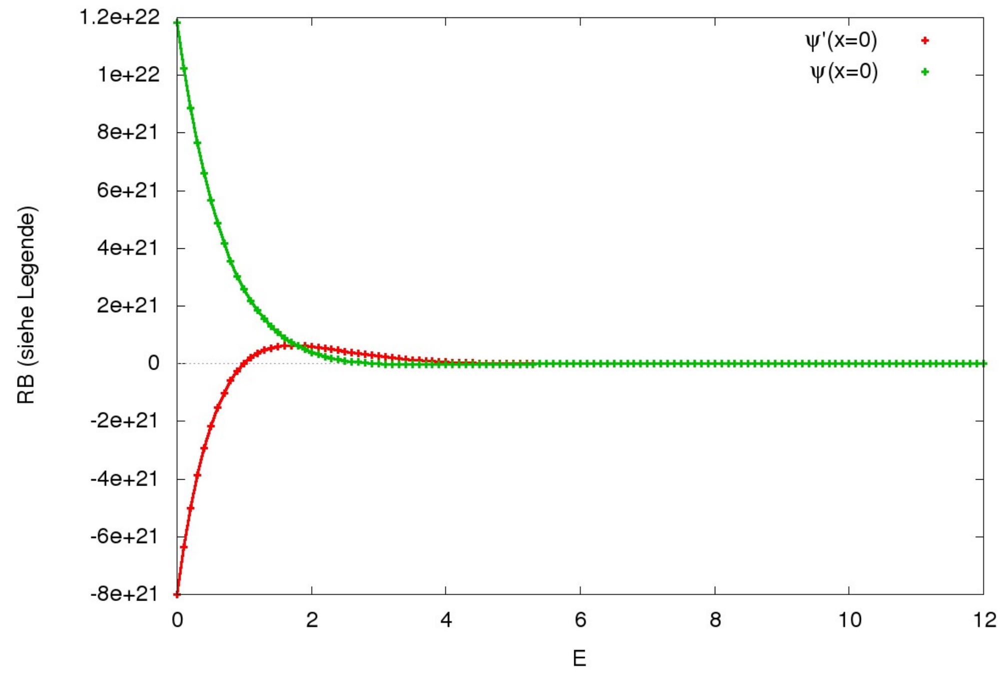


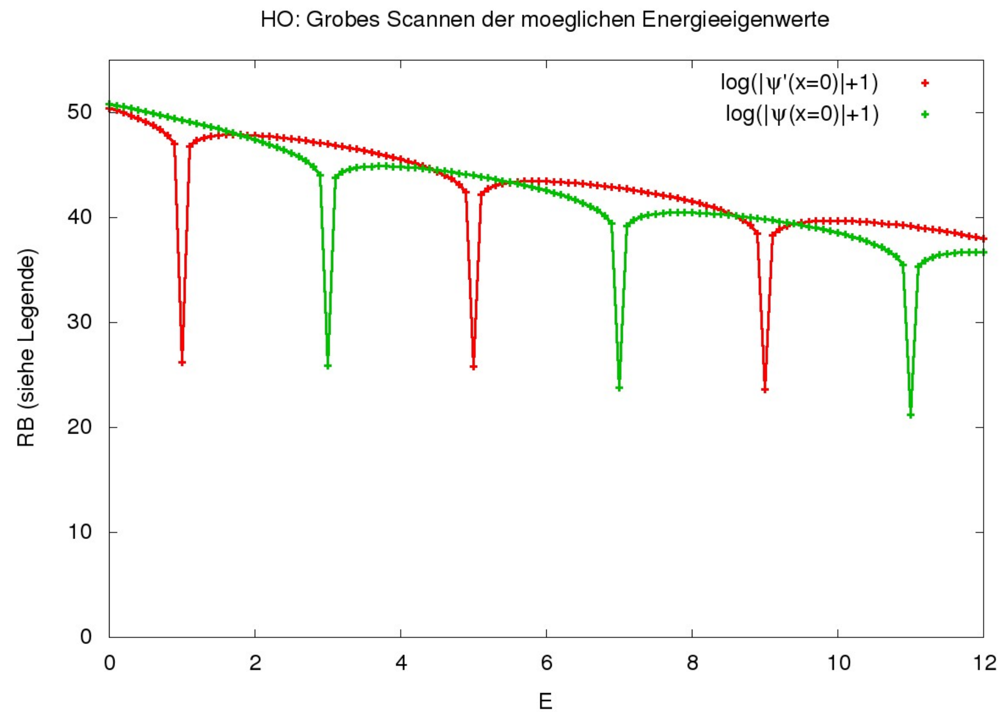
- **Zweckmäßigeres Vorgehen:** Nutze obige Eigenschaft aus ... starte weit im klassisch verbotenen Bereich mit "beliebigen" ABs, z.B.

$\psi(L/a) = 1$,
$\phi(L/a) = 0$,
$\hat{E}(L/a) =?$
oder
$\psi(L/a) = 0$,
$\phi(L/a) = 1$,
$\hat{E}(L/a) =?$

... verwende Shooting, um $\psi'(0) = 0$ (bzw. $\psi(0) = 0$) durch Einstellen von $\hat{E}$ zu realisieren:

- Grobes Scannen der der möglichen Energieeigenwerte: Im Bereich $0.0 \leq \hat{E} \leq 100.0$ liegen drei Energieeigenwerte, $\hat{E}_1 \approx 10.0$, $\hat{E}_1 \approx 40.0$, $\hat{E}_1 \approx 90.0$.


HO: Grobes Scannen der moeglichen Energieeigenwerte
ψ'(x=0)
ψ(x=0)
RB (siehe Legende)
E
1.2e+22
1e+22
8e+21
6e+21
4e+21
2e+21
0
-2e+21
-4e+21
-6e+21
-8e+21
0
2
4
6
8
10
12



- Newton-Raphson-Iteration:
  - Grundzustand:

```
E_num =  +0.900000 .
E_num =  +0.988598 .
E_num =  +0.999834 .
E_num =  +1.000000 .
```

  - 1. angeregter Zustand:

```
E_num =  +2.900000 .
E_num =  +2.988617 .
```

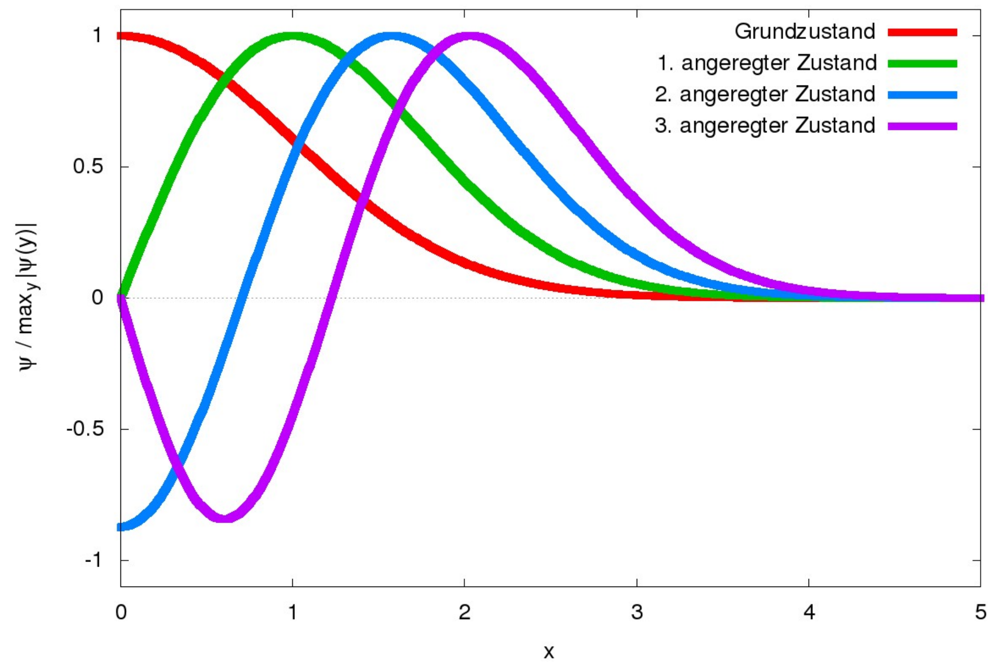
```
E_num =  +2.999835 .
E_num =  +3.000000 .
```

- 2. angeregter Zustand:

```
E_num =  +4.900000 .
E_num =  +4.990699 .
E_num =  +4.999911 .
E_num =  +5.000000 .
```

- 3. angeregter Zustand:

```
E_num =  +6.900000 .
E_num =  +6.990720 .
E_num =  +6.999911 .
E_num =  +7.000000 .
```


HO: (Nicht-normierte) Wellenfunktionen der niedrigsten Energieeigenzustaende
Grundzustand
1. angeregter Zustand
2. angeregter Zustand
3. angeregter Zustand
ψ / max_y |ψ(y)|
x
1
0.5
0
-0.5
-1
0
1
2
3
4
5