Numerische Methoden der Physik

3 Gewöhnliche Differentialgleichungen, Anfangswertprobleme

Marc Wagner

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

SS 2014

3.3 Runge-Kutta-Methode

- 1D harmonischer Oszillator:
  - Newtonsche Bewegungsgleichung ist eine Differentialgleichung 2. Ordnung:

$$m\ddot{x}(t) \quad = \quad -m\omega^2 x(t).$$

  - Umschreiben in ein System zweier Differentialgleichungen 1. Ordnung:

$$\dot{\mathbf{y}}(t) \; = \; \mathbf{f}(\mathbf{y}(t), t) \quad , \quad \mathbf{y}(t) \; = \; (x(t), v(t)) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) \; = \; (v(t), -\omega^2 x(t)).$$

- **C**-Programm:

```
// RK.C

// Loese DGl-System
//
//   \vec{\dot{y}}(t) = \vec{f}(\vec{y}(t),t)
//
// mit Anfangsbedingungen
//
//   \vec{y}(t=0) = \vec{y}_0 .
//
// Hier speziell fuer den harmonischen Oszillator mit Potential
```

```
12. //
13. //   V(x) = m \omega^2 x^2 / 2 .
14.
15. // **********
16.
17. #define __EULER__
18. // #define __RK_2ND__
19. // #define __RK_3RD__
20. // #define __RK_4TH__
21.
22. // **********
23.
24. #include <math.h>
25. #include <stdio.h>
26. #include <stdlib.h>
27.
28. // **********
29.
30. const int N = 2;  // Anzahl der Komponenten von \vec{y} bzw. \vec{f}.
31.
32. const double omega = 1.0;  // Frequenz.
33.
34. const int num_steps = 10000;  // Anzahl der RK-Schritte.
35. const double tau = 0.1;  // Schrittweite.
36.
37. // **********
38.
```

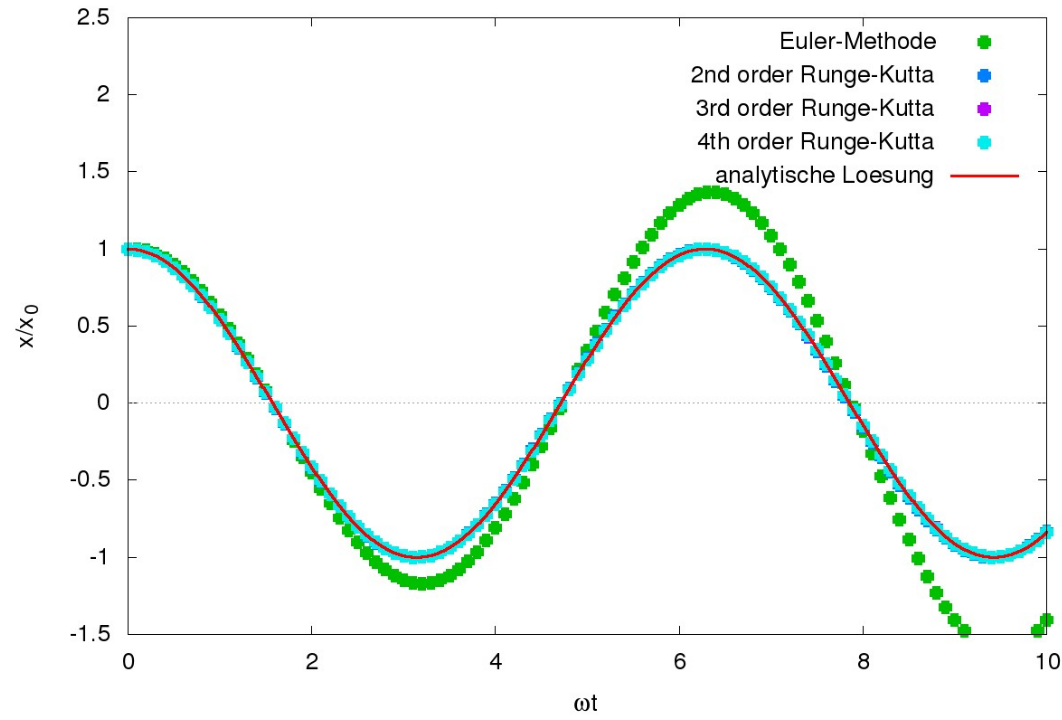
```
39. double y[N][num_steps+1];  // Diskretisierte "Bahnkurven".
40.
41. double y_0[N] = { 1.0 , 0.0 };  // Anfangsbedingungen.
42.
43. // **********
44.
45. int main(int argc, char **argv)
46. {
47.   int i1, i2;
48.
49.   // *****
50.
51.   // Initialisiere "Bahnkurven" mit Anfangsbedingungen.
52.
53.   for(i1 = 0; i1 < N; i1++)
54.     y[i1][0] = y_0[i1];
55.
56.   // *****
57.
58.   // Fuehre Euler/RK-Schritte aus.
59.
60.   for(i1 = 1; i1 <= num_steps; i1++)
61.     {
62.       // 1D-HO:
63.       // y(t) = (x(t) , \dot{x}(t)) ,
64.       // \dot{y}(t) = f(y(t),t) = (\dot{x}(t) , F/m)
65.       //   wobei Kraft F = -m \omega^2 x(t) .
```

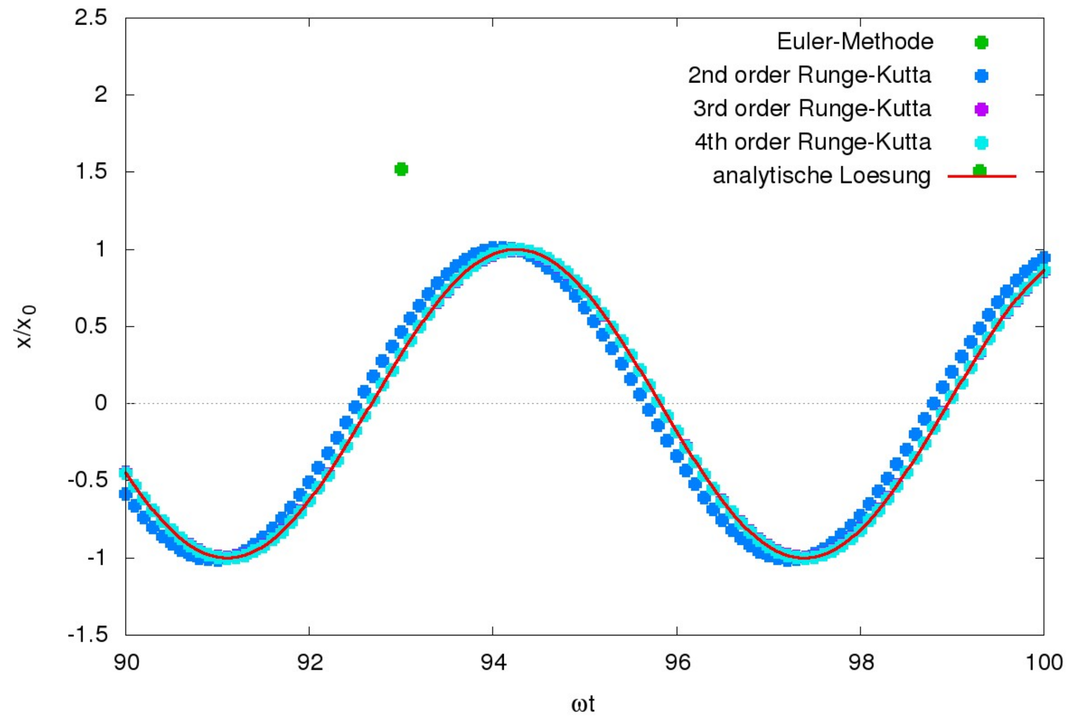
```
66.
67. #ifdef __EULER__
68.
69.       // Berechne k1 = f(y(t),t) * tau.
70.
71.       double k1[N];
72.
73.       k1[0] = y[1][i1-1]    * tau;
74.       k1[1] = -pow(omega, 2.0) * y[0][i1-1]    * tau;
75.
76.       // *****
77.
78.       for(i2 = 0; i2 < N; i2++)
79.         y[i2][i1] = y[i2][i1-1] + k1[i2];
80.
81. #endif
82.
83. #ifdef __RK_2ND__
84.
85.       // Berechne k1 = f(y(t),t) * tau.
86.
87.       double k1[N];
88.
89.       k1[0] = y[1][i1-1]    * tau;
90.       k1[1] = -pow(omega, 2.0) * y[0][i1-1]    * tau;
91.
92.       // *****
```

```
 93.
 94.       // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.
 95.
 96.       double k2[N];
 97.
 98.       k2[0] = (y[1][i1-1] + 0.5*k1[1])   * tau;
 99.       k2[1] = -pow(omega, 2.0) * (y[0][i1-1] + 0.5*k1[0])   * tau;
100.
101.       // *****
102.
103.       for(i2 = 0; i2 < N; i2++)
104.         y[i2][i1] = y[i2][i1-1] + k2[i2];
105.
106. #endif
107.
108. #ifdef __RK_3RD__
109.
110. ...
111.
112. #endif
113.
114. #ifdef __RK_4TH__
115.
116. ...
117.
118. #endif
119.     }
```

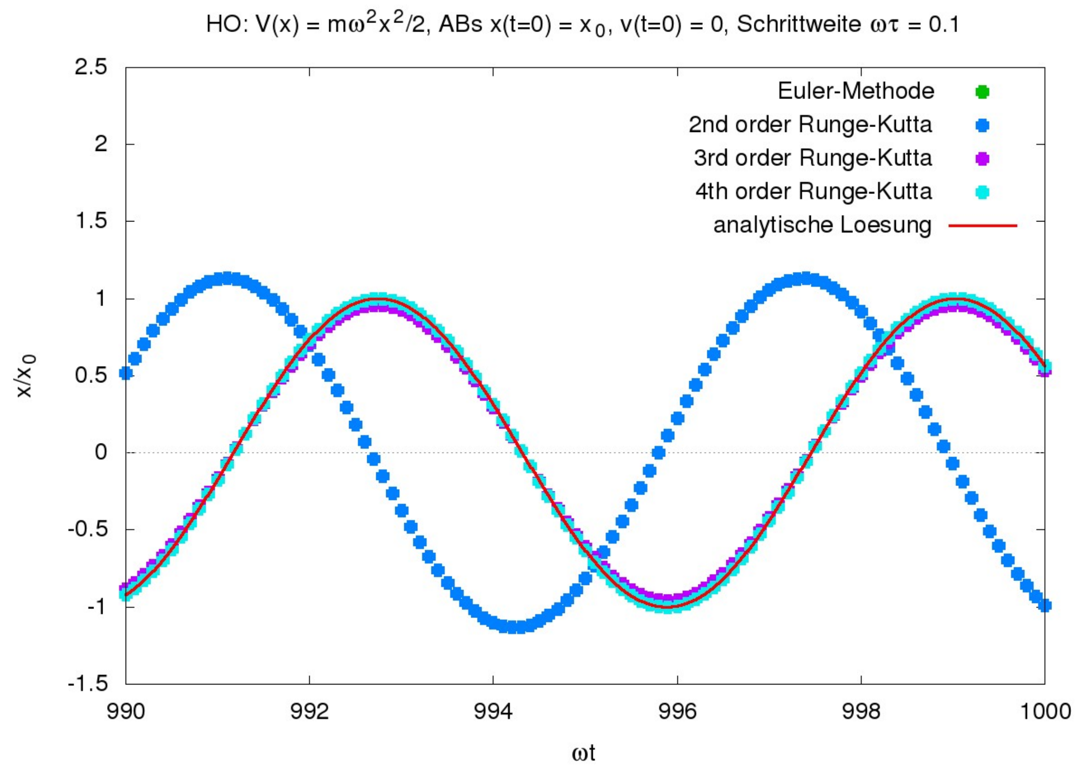
```
120.
121.   // *****
122.
123.   // Ausgabe.
124.
125.   for(i1 = 0; i1 <= num_steps; i1++)
126.     {
127.       double t = i1 * tau;
128.       printf("%9.6lf %9.6lf %9.6lf\n", t, y[0][i1], y[0][i1]-cos(t));
129.     }
130.
131.   // *****
132.
133.   return EXIT_SUCCESS;
134. }
```

- Vergleich von Euler, 2nd, 3rd und 4th order Runge-Kutta:

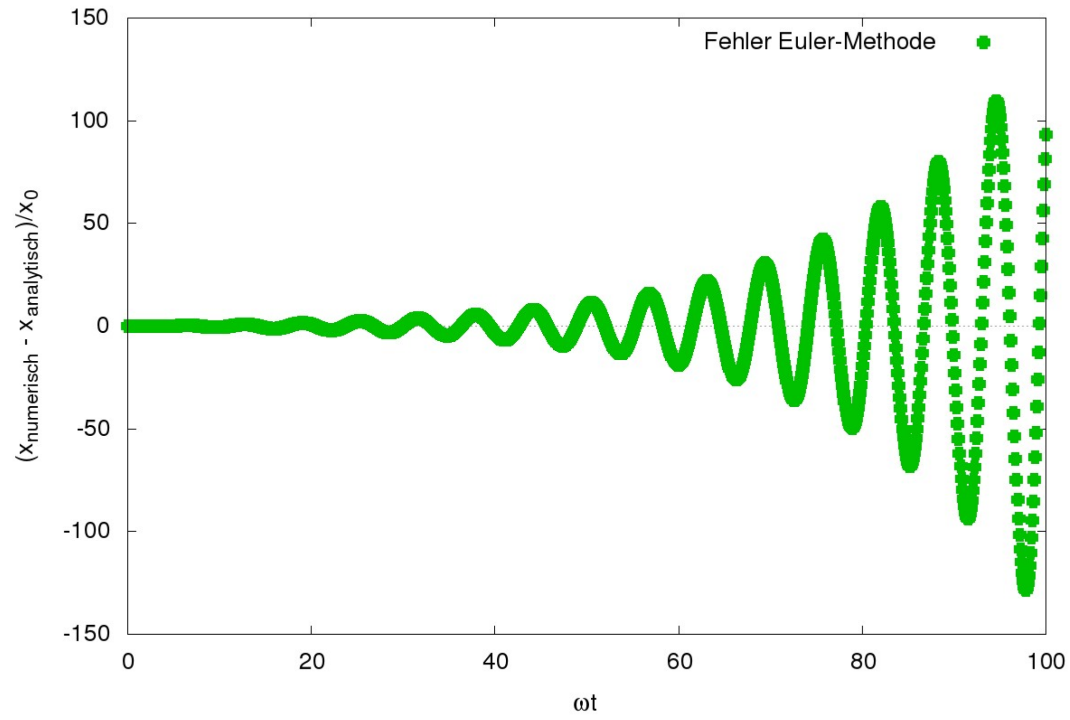

HO: V(x) = mω²x²/2, ABs x(t=0) = x₀, v(t=0) = 0, Schrittweite ωτ = 0.1
Euler-Methode
2nd order Runge-Kutta
3rd order Runge-Kutta
4th order Runge-Kutta
analytische Loesung
x/x₀
ωt
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10



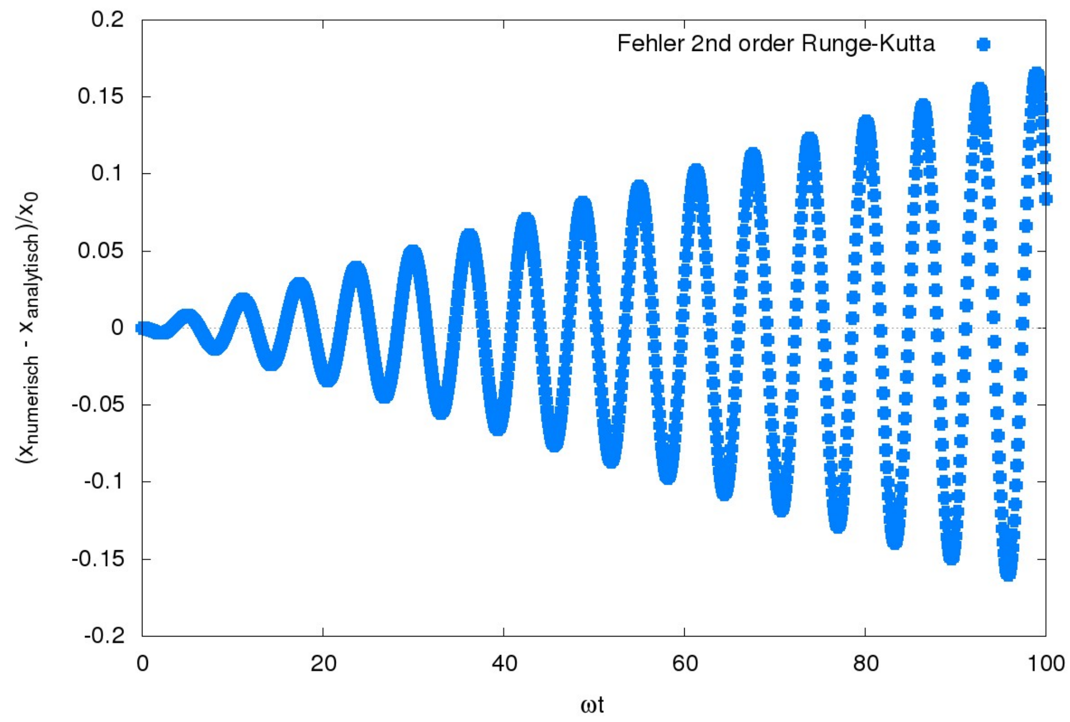
HO: V(x) = mω²x²/2, ABs x(t=0) = x0, v(t=0) = 0, Schrittweite ωτ = 0.1
Euler-Methode
2nd order Runge-Kutta
3rd order Runge-Kutta
4th order Runge-Kutta
analytische Loesung
x/x0
ωt
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
90
92
94
96
98
100



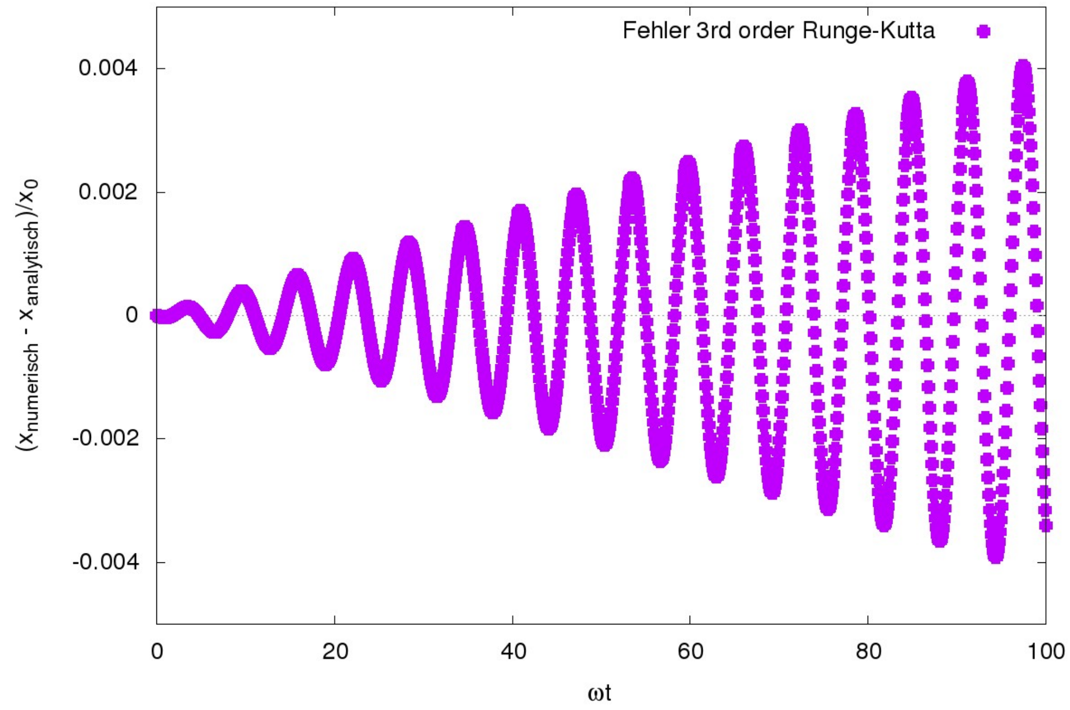
- Fehler von Euler, 2nd, 3rd und 4th order Runge-Kutta:



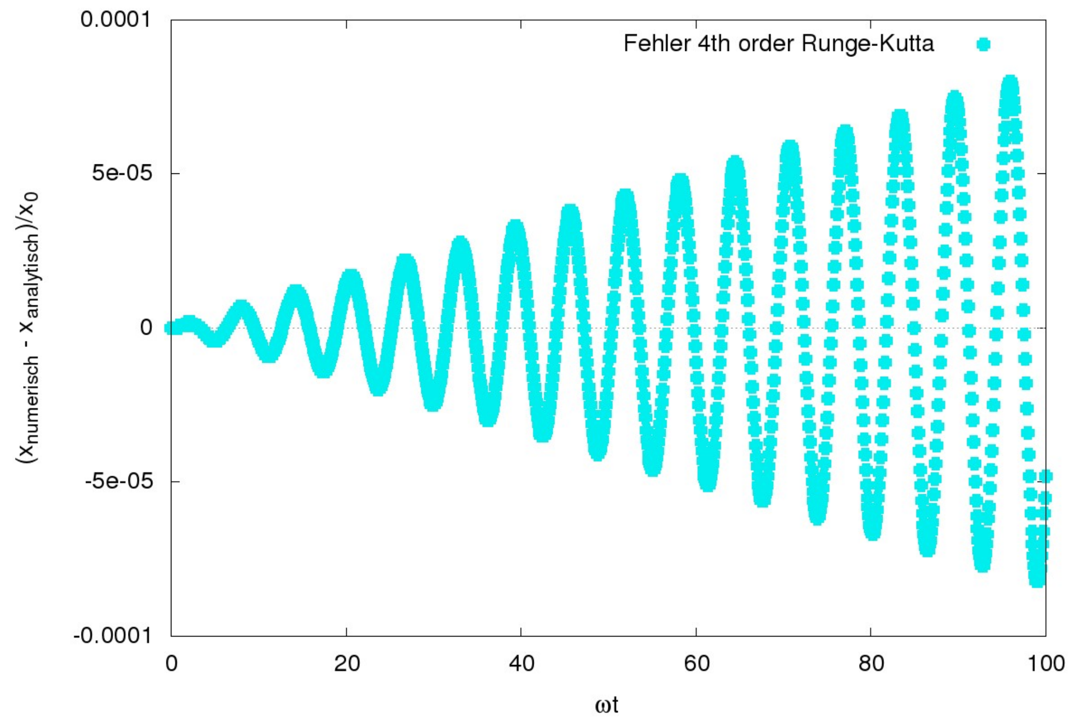
HO: V(x) = mω²x²/2, ABs x(t=0) = x₀, v(t=0) = 0, Schrittweite ωτ = 0.1
Fehler Euler-Methode
(x_numerisch - x_analytisch)/x_0
ωt
150
100
50
0
-50
-100
-150
0
20
40
60
80
100



HO: V(x) = mω²x²/2, ABs x(t=0) = x₀, v(t=0) = 0, Schrittweite ωτ = 0.1
Fehler 2nd order Runge-Kutta
(x_numerisch - x_analytisch)/x_0
ωt
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
0
20
40
60
80
100



HO: V(x) = mω²x²/2, ABs x(t=0) = x 0, v(t=0) = 0, Schrittweite ωτ = 0.1
Fehler 3rd order Runge-Kutta
(x numerisch - x analytisch)/x 0
0.004
0.002
0
-0.002
-0.004
0
20
40
60
80
100
ωt


HO: V(x) = mω²x²/2, ABs x(t=0) = x 0, v(t=0) = 0, Schrittweite ωτ = 0.1
Fehler 4th order Runge-Kutta
(x numerisch - x analytisch)/x 0
ωt
0.0001
5e-05
0
-5e-05
-0.0001
0
20
40
60
80
100

3.3.2 Dynamische Anpassung der Schrittweite

- 1D anharmonischer Oszillator:
  - Newtonsche Bewegungsgleichung ist eine Differentialgleichung 2. Ordnung:

$$m\ddot{x}(t) \quad = \quad -\alpha n(x(t))^{n-1}.$$

  - Umschreiben in ein System zweier Differentialgleichungen 1. Ordnung:

$$\dot{\mathbf{y}}(t) \quad = \quad \mathbf{f}(\mathbf{y}(t), t) \quad , \quad \mathbf{y}(t) \quad = \quad (x(t), v(t)) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) \quad = \quad (v(t), -\alpha n(x(t))^{n-1}/m)$$

- **C**-Programm:

```
1. // **********
2. 
3. 
4. 
5. // RK_dynamical_stepsize.C
6. 
7. // Loese DGl-System
8. //
9. //   \vec{\dot{y}}(t) = \vec{f}(\vec{y}(t),t)
10. //
11. // mit Anfangsbedingungen
```

```
12. //
13. //   \vec{y}(t=0) = \vec{y}_0 .
14. //
15. // Hier speziell fuer den anharmonischen Oszillator mit Potential
16. //
17. //   V(x) = \alpha x^n
18. //
19. // Mit dynamischer Schrittweitenanpassung.
20.
21.
22.
23. // **********
24.
25.
26.
27. #include <math.h>
28. #include <stdio.h>
29. #include <stdlib.h>
30.
31.
32.
33. // **********
34.
35.
36.
37. // ********************
38. // Physikalische Parameter, etc.
```

```
39. // ********************
40.
41.
42. // Anharmonischer Oszillator, V(x) = \alpha x^n,
43. //   y = (x , v)
44. //   f = (v , -\alpha n x^n-1/m) .
45.
46.
47. const int N = 2;  // Anzahl der Komponenten von y bzw. f.
48.
49.
50. // const int n = 2;
51. // const double alpha = 0.5;  // \alpha/m
52. const int n = 20;
53. const double alpha = 1.0;  // \alpha/m
54.
55. double y_0[N] = { 1.0 , 0.0 };  // Anfangsbedingungen y(t=0).
56.
57.
58. // Berechnet f(y(t),t) * tau.
59.
60. void f_times_tau(double *y_t, double t, double *f_times_tau_, double tau)
61. {
62.   if(N != 2)
63.     {
64.       fprintf(stderr, "Error: N != 2!\n");
65.       exit(EXIT_FAILURE);
```

```
66.     }
67.
68.   f_times_tau_[0] = y_t[1] * tau;
69.   f_times_tau_[1] = -alpha * ((double)n) * pow(y_t[0], ((double)(n-1))) * tau;
70. }
71.
72.
73.
74. // **********
75.
76.
77.
78. // ********************
79. // RK-Parameter.
80. // ********************
81.
82.
83. // #define __EULER__
84. #define __RK_2ND__
85. // #define __RK_3RD__
86. // #define __RK_4TH__
87.
88. #ifdef __EULER__
89. const int order = 1;
90. #endif
91.
92. #ifdef __RK_2ND__
```

```
 93. const int order = 2;
 94. #endif
 95.
 96. #ifdef __RK_3RD__
 97. const int order = 3;
 98. #endif
 99.
100. #ifdef __RK_4TH__
101. const int order = 4;
102. #endif
103.
104. // Maximale Anzahl der RK-Schritte.
105. const int num_steps_max = 10000;
106.
107. // Sobald t >= t_max wird abgebrochen.
108. const double t_max = 10.0;
109.
110. // Maximal zulaessiger Fehler pro Schritt.
111. const double delta_abs_max = 0.001;
112.
113. double tau = 1.0;  // Initiale (grobe) Schrittweite.
114.
115.
116.
117. // **********
118.
119.
```

```
120.
121. double t[num_steps_max+1];  // Diskretisierte Zeitachse.
122. double y[num_steps_max+1][N];  // Diskretisierte "Bahnkurven".
123.
124.
125.
126. // **********
127.
128.
129.
130. #ifdef __EULER__
131.
132. ...
133.
134. #endif
135.
136.
137.
138. #ifdef __RK_2ND__
139.
140. // RK-Schritt (2nd order) um Schrittweite tau.
141.
142. void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
143. {
144.   int i1;
145.
146.   // *****
```

```
147.
148.   // Berechne k1 = f(y(t),t) * tau.
149.
150.   double k1[N];
151.   f_times_tau(y_t, t, k1, tau);
152.
153.   // *****
154.
155.   // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.
156.
157.   double y_[N];
158.
159.   for(i1 = 0; i1 < N; i1++)
160.     y_[i1] = y_t[i1] + 0.5*k1[i1];
161.
162.   double k2[N];
163.   f_times_tau(y_, t + 0.5*tau, k2, tau);
164.
165.   // *****
166.
167.   for(i1 = 0; i1 < N; i1++)
168.     y_t_plus_tau[i1] = y_t[i1] + k2[i1];
169. }
170.
171. #endif
172.
173.
```

```
174.
175. #ifdef __RK_3RD__
176.
177. ...
178.
179. #endif
180.
181.
182.
183. #ifdef __RK_4TH__
184.
185. ...
186.
187. #endif
188.
189.
190.
191. // **********
192.
193.
194.
195. int main(int argc, char **argv)
196. {
197.   double d1;
198.   int i1, i2;
199.
200.   // *****
```

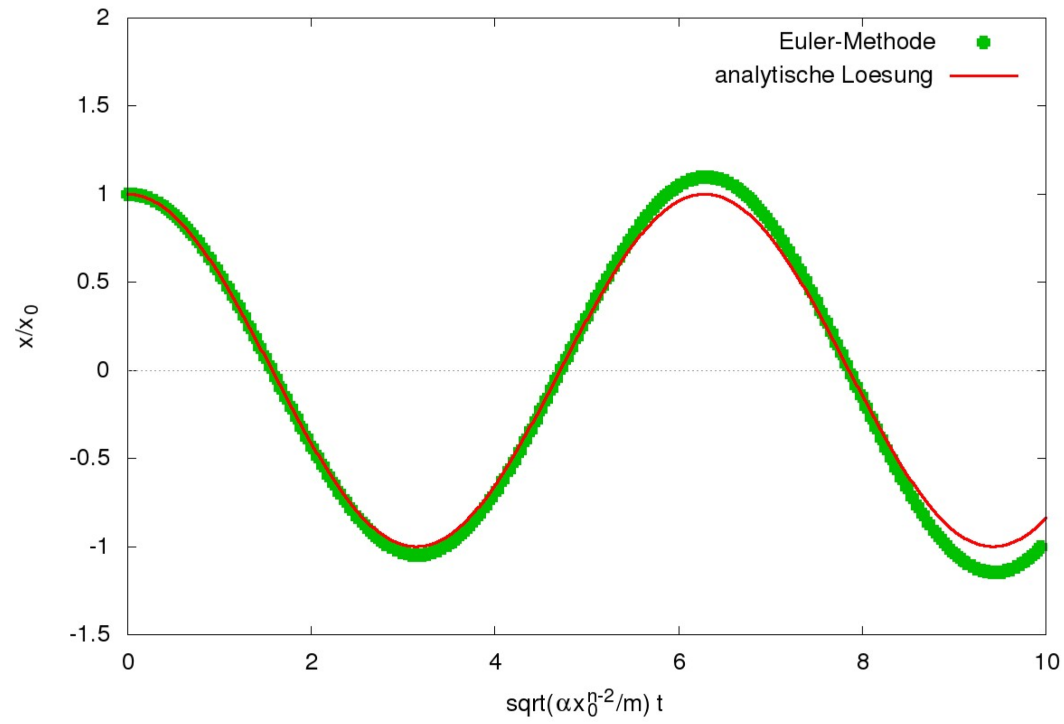
```
201.
202.   // Initialisiere "Bahnkurven" mit Anfangsbedingungen.
203.
204.   t[0] = 0.0;
205.
206.   for(i1 = 0; i1 < N; i1++)
207.     y[0][i1] = y_0[i1];
208.
209.   // *****
210.
211.   // Fuehre Euler/RK-Schritte aus.
212.
213.   for(i1 = 0; i1 < num_steps_max; i1++)
214.     {
215.       if(t[i1] >= t_max)
216.         break;
217.
218.       // *****
219.
220.       // RK-Schritte.
221.
222.       double y_tau[N], y_tmp[N], y_2_x_tau_over_2[N];
223.
224.       // y(t)   --> \tau   y_{\tau}(t+\tau)
225.       RK_step(y[i1], t[i1], y_tau, tau);
226.
227.       // y(t)   --> \tau/2   --> \tau_2   y_{2 * \tau / 2}(t+\tau)
```

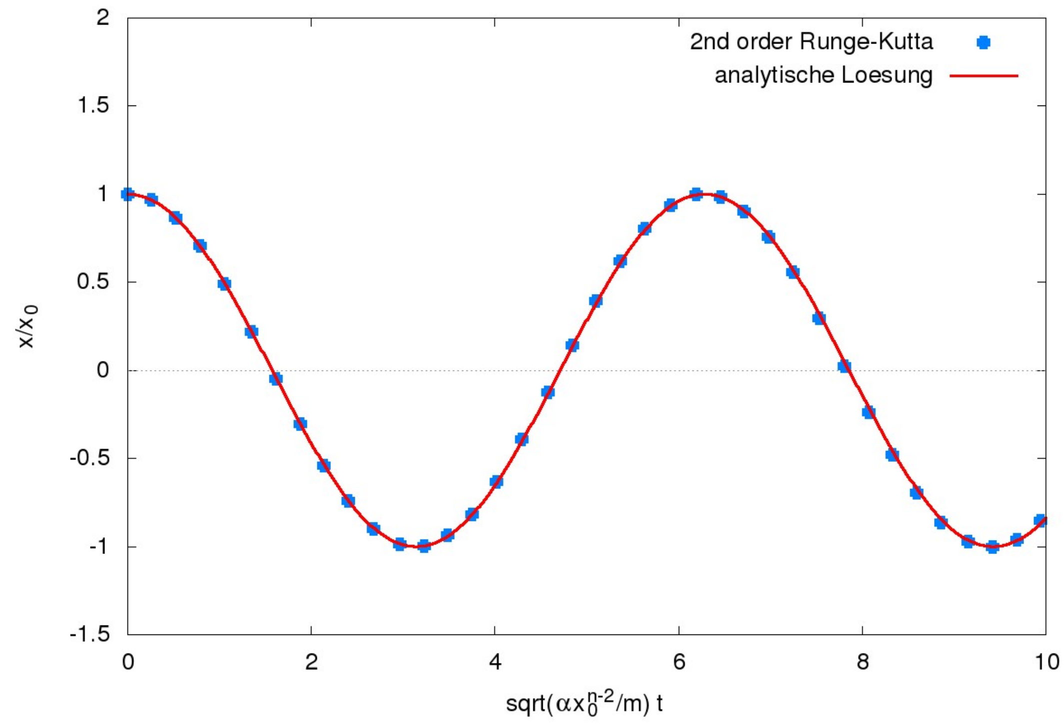
```
228.      RK_step(y[i1], t[i1], y_tmp, 0.5*tau);
229.      RK_step(y_tmp, t[i1]+0.5*tau, y_2_x_tau_over_2, 0.5*tau);
230.
231.      // *****
232.
233.      // Fehlerabschaetzung.
234.
235.      double delta_abs = fabs(y_2_x_tau_over_2[0] - y_tau[0]);
236.
237.      for(i2 = 1; i2 < N; i2++)
238.        {
239.          d1 = fabs(y_2_x_tau_over_2[i2] - y_tau[i2]);
240.
241.          if(d1 > delta_abs)
242.            delta_abs = d1;
243.        }
244.
245.      delta_abs /= pow(2.0, (double)order) - 1.0;
246.
247.      // *****
248.
249.      // Schrittweitenanpassung (maximale Veraenderung um Faktor 5.0).
250.
251.      d1 = 0.9 * pow(delta_abs_max / delta_abs, 1.0 / (((double)order)+1.0));
252.
253.      if(d1 < 0.2)
254.        d1 = 0.2;
```

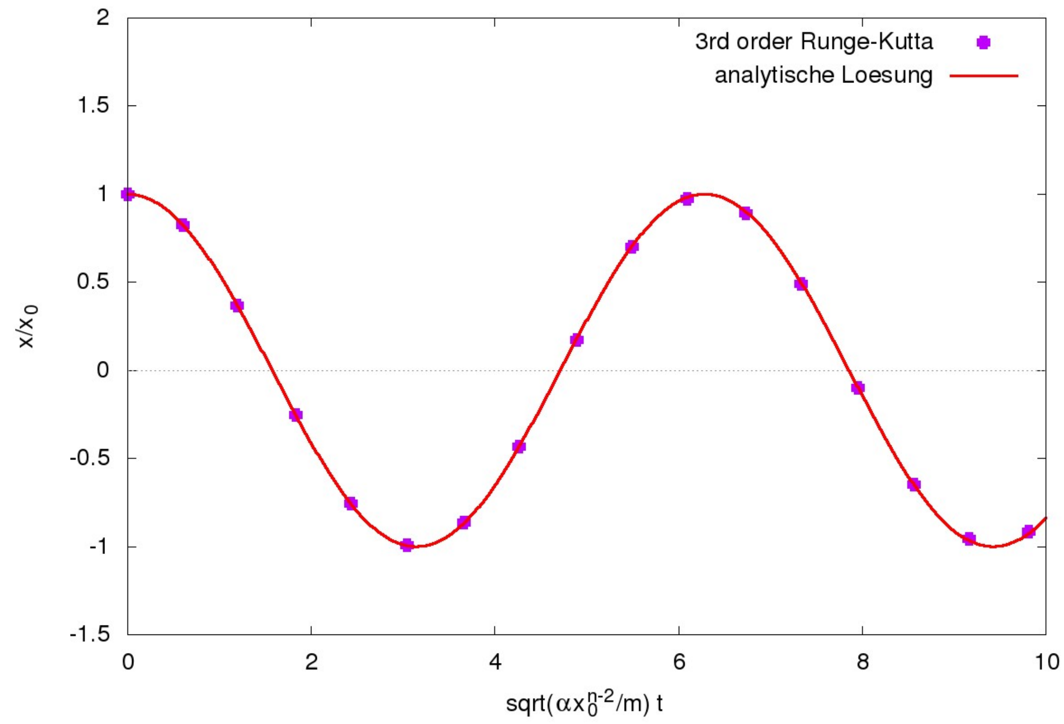
```
255.
256.       if(d1 > 5.0)
257.         d1 = 5.0;
258.
259.       double tau_new = d1 * tau;
260.
261.       // *****
262.
263.       if(delta_abs <= delta_abs_max)
264.         {
265.           // Akzeptieren des RK-Schrittes.
266.
267.           for(i2 = 0; i2 < N; i2++)
268.             y[i1+1][i2] = y_2_x_tau_over_2[i2];
269.
270.           t[i1+1] = t[i1] + tau;
271.
272.           tau = tau_new;
273.         }
274.       else
275.         // Wiederholen des RK-Schrittes mit kleinerer Schrittweite.
276.         {
277.           tau = tau_new;
278.
279.           i1--;
280.         }
281.     }
```

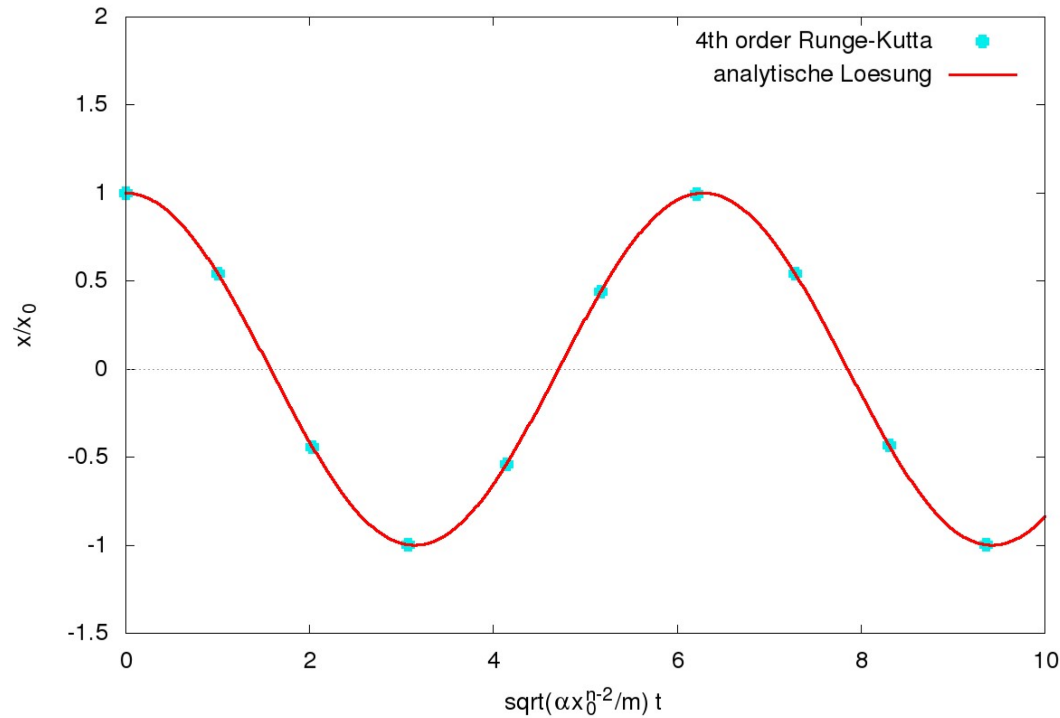
```
282.
283.   int num_steps = i1;
284.
285.   // *****
286.
287.   // Ausgabe.
288.
289.   for(i1 = 0; i1 <= num_steps; i1++)
290.     {
291.       printf("%9.6lf %9.6lf\n", t[i1], y[i1][0]);
292.     }
293.
294.   // *****
295.
296.   return EXIT_SUCCESS;
297. }
298.
299.
300.
301. // **********
```

- Vergleich von Euler, 2nd, 3rd und 4th order Runge-Kutta (dynamische Schrittweitenanpassung, $V(x) = x^2/2$ (harmonischer Oszillator)):

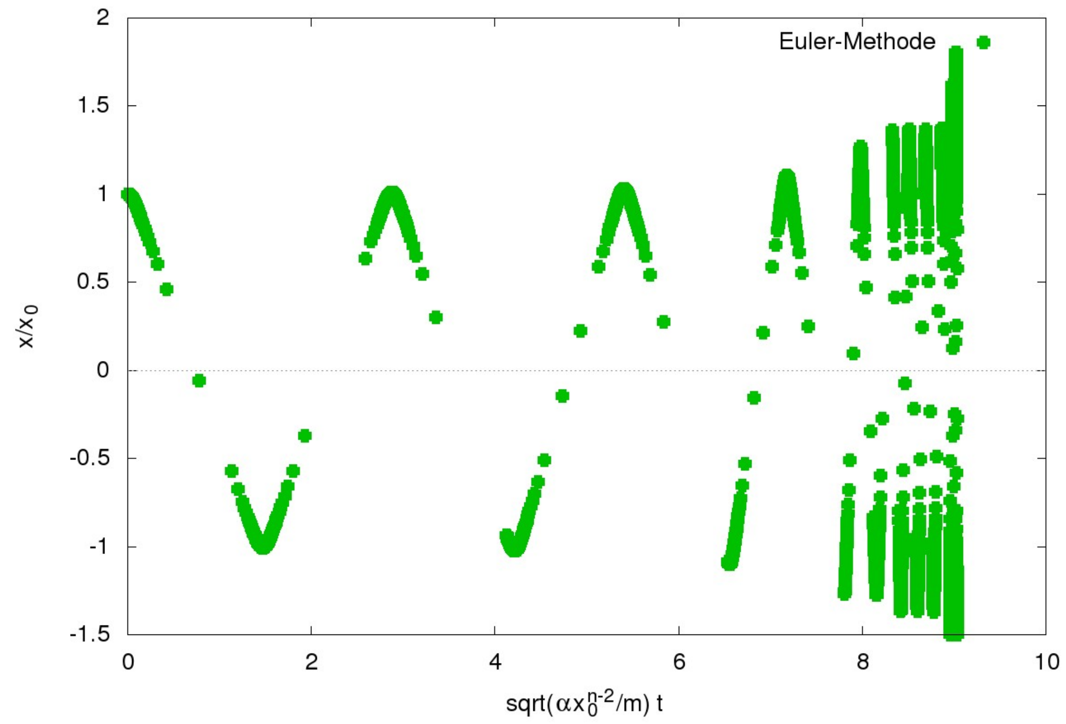

AHO: V(x) = αx^n, n = 2, α = 0.5, ABs x(t=0) = x0, v(t=0) = 0, δabs,max = 0.001, τinitial = 1.0
Euler-Methode
analytische Loesung
x/x0
sqrt(αx0^(n-2)/m) t
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10


AHO: V(x) = αx^n, n = 2, α = 0.5, ABs x(t=0) = x_0, v(t=0) = 0, δ_abs,max = 0.001, τ_initial = 1.0
2nd order Runge-Kutta
analytische Loesung
x/x_0
sqrt(αx_0^n-2/m) t
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10

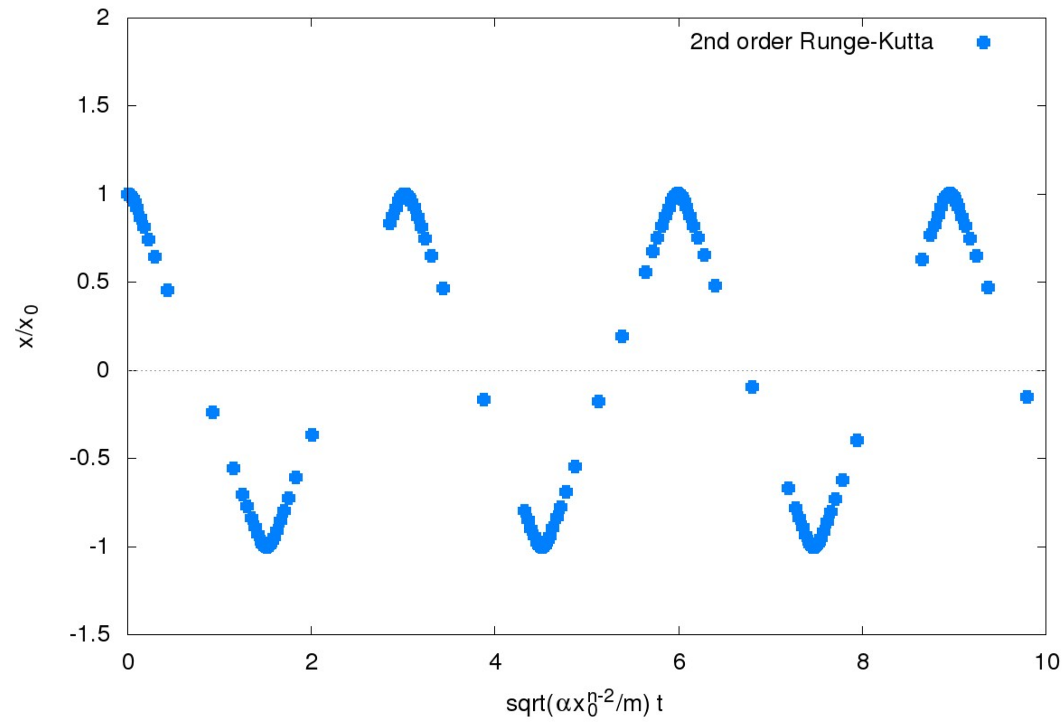

AHO: V(x) = αx^n, n = 2, α = 0.5, ABs x(t=0) = x_0, v(t=0) = 0, δ_abs,max = 0.001, τ_initial = 1.0
3rd order Runge-Kutta
analytische Loesung
x/x_0
sqrt(αx_0^{n-2}/m) t
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10



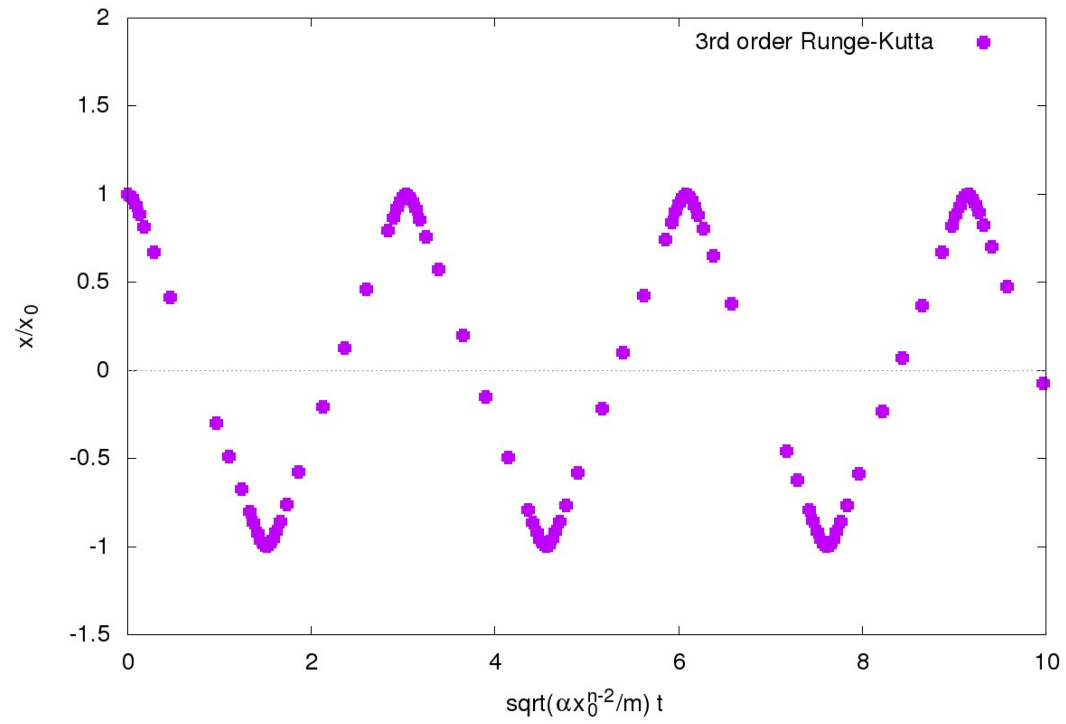
- Vergleich von Euler, 2nd, 3rd und 4th order Runge-Kutta (dynamische Schrittweitenanpassung, $V(x) = x^{20}$ (anharmonischer Oszillator)):

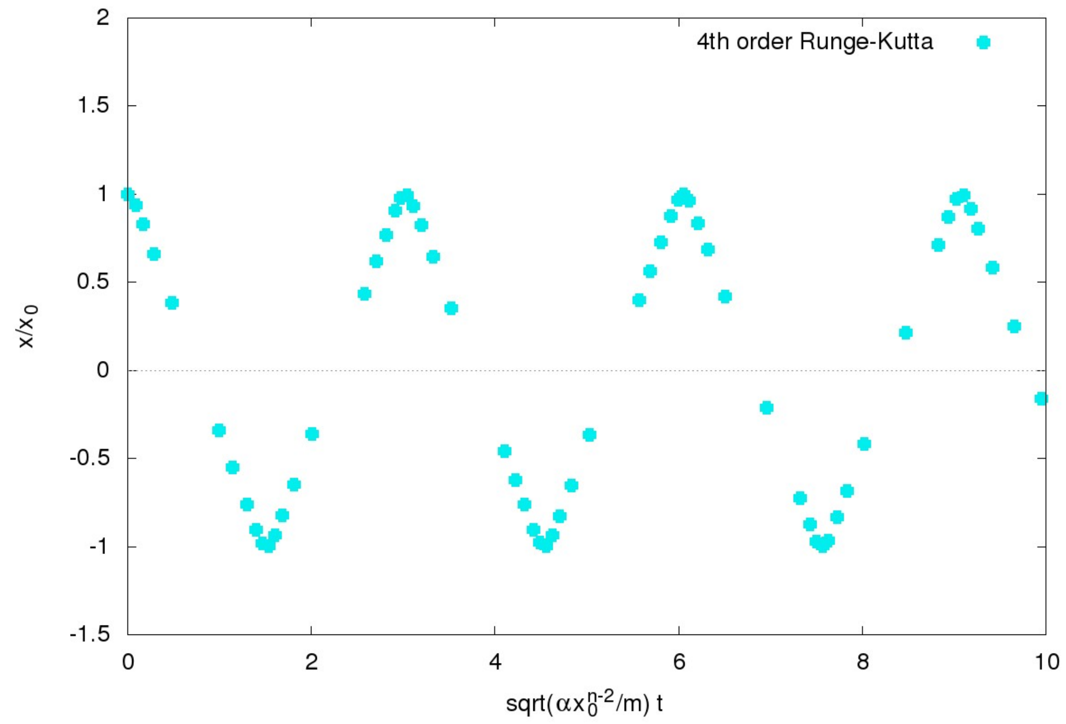


AHO: V(x) = αx^n, n = 20, α = 1.0, ABs x(t=0) = x_0, v(t=0) = 0, δ_abs,max = 0.001, τ_initial = 1.0
Euler-Methode
x/x_0
sqrt(αx_0^(n-2)/m) t
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10



AHO: V(x) = αx^n, n = 20, α = 1.0, ABs x(t=0) = x_0, v(t=0) = 0, δ_abs,max = 0.001, τ_initial = 1.0
2nd order Runge-Kutta
x/x_0
sqrt(αx_0^(n-2)/m) t
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10


AHO: V(x) = αx^n, n = 20, α = 1.0, ABs x(t=0) = x_0, v(t=0) = 0, δ_abs,max = 0.001, τ_initial = 1.0
3rd order Runge-Kutta
x/x_0
sqrt(αx_0^(n-2)/m) t
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10


AHO: V(x) = αx^n, n = 20, α = 1.0, ABs x(t=0) = x_0, v(t=0) = 0, δ_abs,max = 0.001, τ_initial = 1.0
4th order Runge-Kutta
x/x_0
sqrt(αx_0^(n-2)/m) t
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10