

# Computerphysik I: Blatt 03

Aurel Müller-Schönau und Leon Oleschko

23. Mai 2022

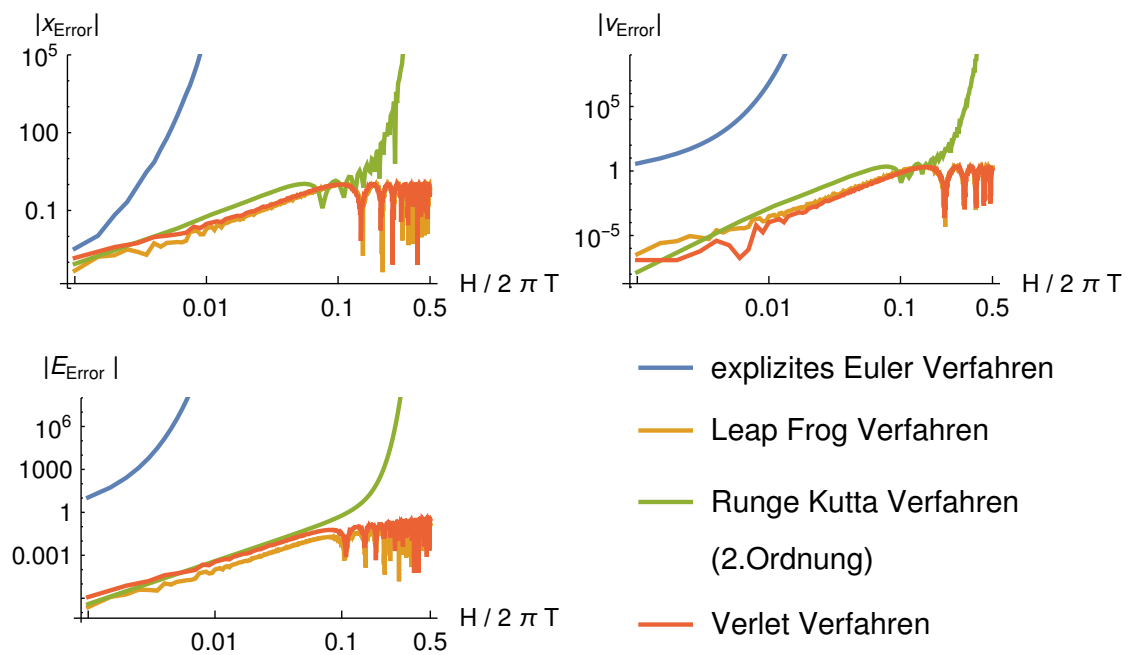


Abbildung 1: Fehler in Abhängigkeit von der Schrittweite  $H$

## a) Fehlerabhängigkeit

Um die Fehlerabhängigkeit von verschiedenen numerischen Methoden zu prüfen, wurde ein harmonischer Oszillator (Periodendauer  $T$ ) für 500 Oszillationen mit verschiedenen Zeitlichen Auflösungen  $H$  simuliert. Dabei ist  $H \in [0.001; 0.001; 0.5]/2\pi T$ . Dies ist zwar unrealistisch hoch, lässt dafür aber eine schnelle Simulation zu.

In der Abbildung 1 sind der relative Auslenkungsfehler  $x$ , der Geschwindigkeitsfehler  $v$  und Energiefehler  $E$  für verschiedene Zeitaufösungen  $H$  dargestellt.

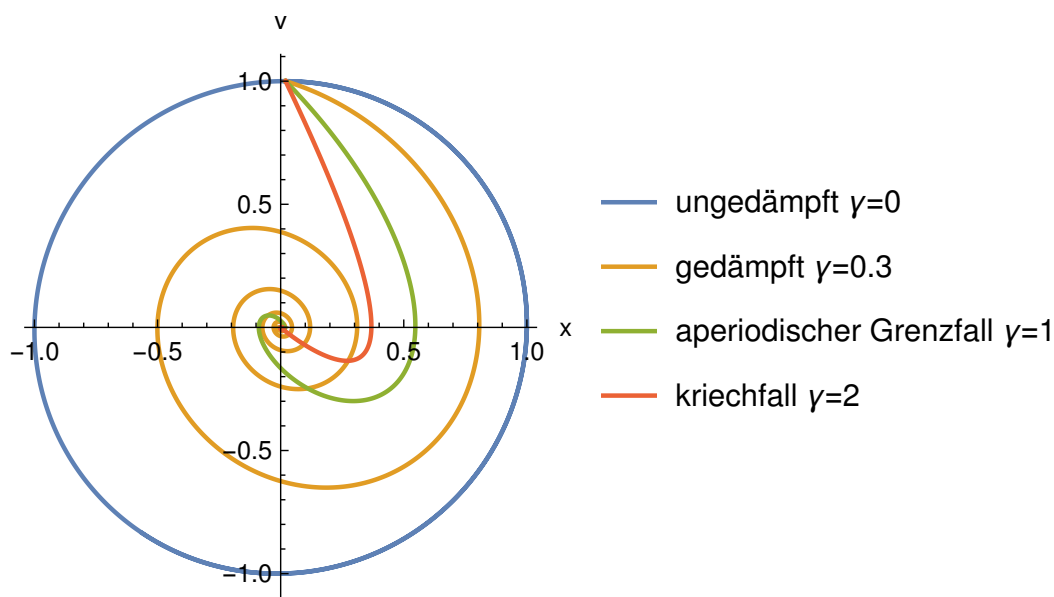
Dass die Energie nur beim Leap-Frog- und Verlet-Verfahren erhalten ist (zumindest beinahe) ist trotzdem gut zu erkennen. (Implementation 2 und 4)

Diese beiden liefern auch nahezu dieselben Ergebnisse, was nicht verwunderlich ist, da beide symmetrische Zweischrittverfahren sind. Die Energie ist erhalten, weil die Schwingung des harmonischen Oszillators unter Zeitumkehr invariant ist, somit ist die Symmetrie der Verfahren hinreichend für Energieerhaltung.

Für kleine Schrittweiten ist das Runge-Kutta-Verfahren beinahe genau so gut wie die symmetrischen Verfahren. Für komplexere Systeme sollte es sogar bessere Ergebnisse liefern aufgrund höherer Konsistenzordnung, jedoch haben die symmetrischen Verfahren in diesem Fall einen systematischen Vorteil. (Implementation 3)

Das explizite Euler-Cauchy-Verfahren schneidet als nicht symmetrisches Einschrittverfahren am schlechtesten ab. Der “Energiegewinn“ ist dadurch zu erklären, dass das Verfahren prinzipbedingt in den Bereichen, in denen die exakte Lösung eine Krümmung besitzt quasi zu weit ausschwenkt, deshalb wird die Schwingungsamplitude immer größer und die Energie wächst immer weiter an. (Implementation 1)

## b) Phasendiagramm



**Abbildung 2:** Phasendiagramm für verschiedene Dämpfungen  $\gamma$

In Abbildung 2 ist das Phasendiagramm für verschiedene Dämpfungen  $\gamma$  dargestellt. Die Ortskoordinate  $x$  ist mit dem Verlet Verfahren in Listing 5 simuliert und die Geschwindigkeit  $v$  wird mit der numerischen rechts Ableitung bestimmt.

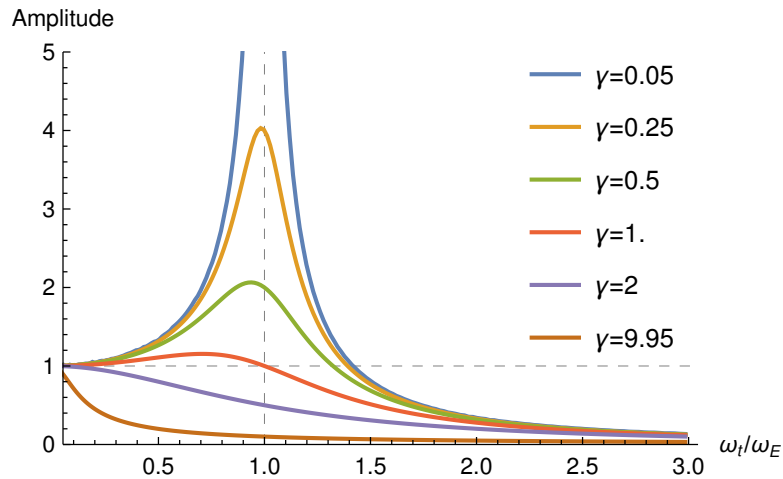


Abbildung 3: Resonanzkurve für 6 verschiedene Dämpfungen  $\gamma$

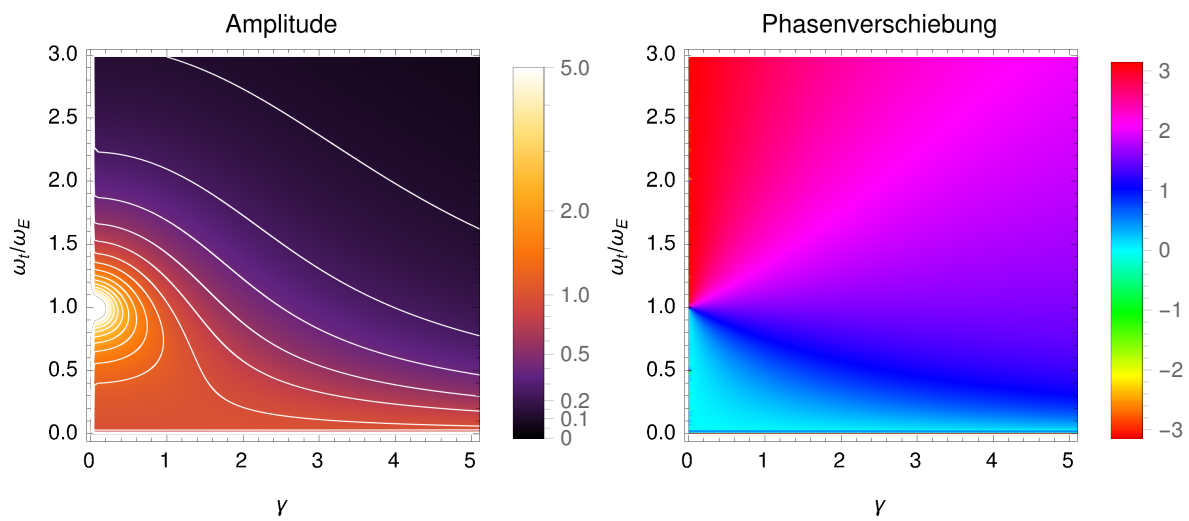
### c,d,e) Angetriebener Oszillator

**c)** In der Abbildung 3 ist die Resonanzkurve eines harmonischen Oszillators mit Eigenfrequenz  $\omega_E$ , mit einer Antreibenden Frequenz  $\omega_t$  und einer Amplitude der Antreibenden Kraft von 1 für 6 verschiedene Dämpfungen  $\gamma$  dargestellt.

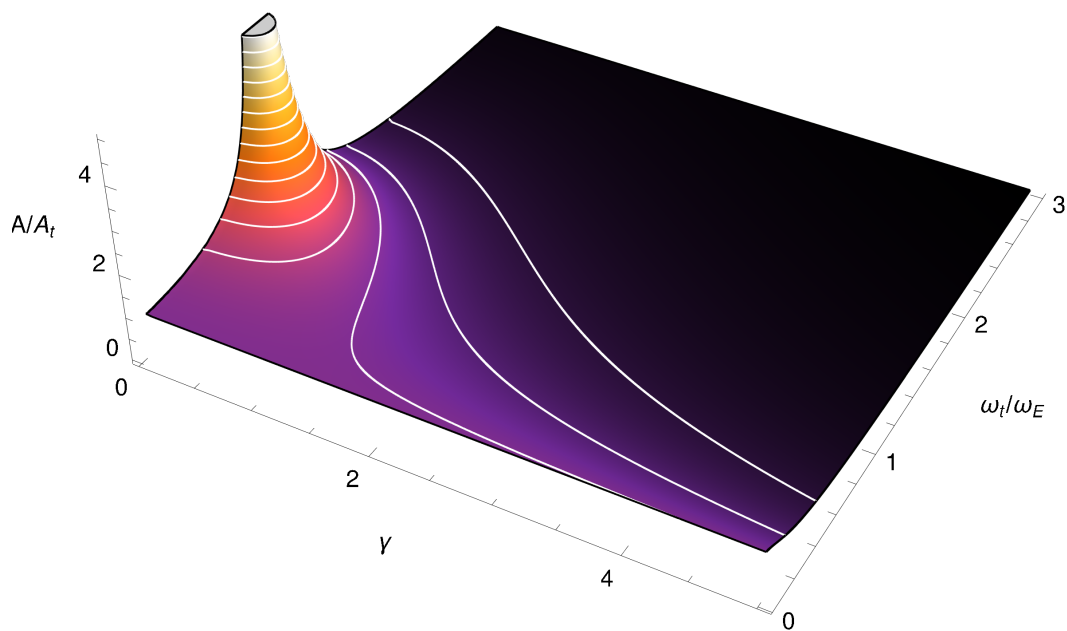
Dabei wird der Ort mit dem Verlet Verfahren (Listing 6) simuliert und die Geschwindigkeit wird mit der numerischen rechts Ableitung bestimmt. Nach einer Einschwingzeit von  $180 \text{ rad } \omega_E^{-1}$  wird die Amplitude bestimmt, indem das Maximum gespeichert wird, bis nach weitere  $120 \text{ rad } \omega_E^{-1}$  das Maximum ausgelesen wird.

**e)** Indem an dem Maximumszeitpunkt die Phase der antreibenden Schwingung  $\omega_T \cdot t \bmod 2\pi$  gespeichert wird, kann auch die Phasenverschiebung zwischen der antreibenden und resultierenden Schwingung bestimmt werden. Diese ist in *rad* in der Abbildung 4 dargestellt. Dabei ist schön der Phasensprung von  $\pi$  bei der Resonanzkatastrophe bei schwacher Dämpfung zu sehen.

**d)** In Abbildung 5 ist das Phasendiagramme für verschiedene Dämpfungen  $\gamma$  als 3 dimensionale Oberfläche dargestellt.



**Abbildung 4:** 2D Resonanzkurve für verschiedene Dämpfungen  $\gamma$  mit Phasenverschiebung zur antreibenden Schwingung in rad.



**Abbildung 5:** 3D Darstellung der Resonanzkurve für verschiedene Dämpfungen  $\gamma$

## Appendix - Code

a)

**Listing 1:** Simulation für die a) mit dem Euler Verfahren für verschiedene Schrittweiten

```

1  /*
2  Solution for a) with the explicit euler method
3  */
4
5  #include <stdio.h>
6  #include <math.h>
7
8  int main(){
9      // define constants
10     const double v_0 = 1., x_0=0., D=1, M=1.;
11     const double T = 1000. * M_PI;
12
13     // loop over step size H
14     for(double H = 0.001; H < 0.5; H+=0.0005){
15         // initialize variables
16         double v = v_0, x = x_0, xneu, vneu, E;
17
18         // simulation loop
19         for(double t = 0; t < T; t += H){
20             // euler step
21             vneu = v - H*D*x/M;
22             xneu = x + H*v;
23
24             // update variables
25             x = xneu;
26             v = vneu;
27
28             // calculate energy
29             E = (M*v*v + D*x*x)/2;
30
31             // print results for debugging
32             //printf("%g %g %g %g\n", t, x, v, E);
33
34         }
35
36         // print difference from analytical solution
37         printf("%g %g %g %g\n", H, x-x_0, v-v_0, E - (M*v_0*v_0 + D*x_0*x_0)
38         /2);
39     }

```

```

39
40 // exit ok
41 return(0);
42 }

```

**Listing 2:** Simulation für die a) mit dem Leap Frog Verfahren für verschiedene Schrittweiten

```

1 /*
2 Solution for a) with the leap frog method
3 */
4
5 #include <stdio.h>
6 #include <math.h>
7
8 int main(){
9 // define constants
10 const double v_0 = 1., x_0=0., D=1, M=1.;
11 const double T = 1000. * M_PI;
12
13 // loop over step size H
14 for(double H = 0.001; H < 0.5; H+=0.0005){
15 // initialize variables
16 double v = v_0, x = x_0, E;
17
18 // simulation loop
19 for(double t = 0; t < T; t += H){
20 // leap frog step
21 x = x + H*v;
22 v = v - H*D*x/M;
23
24 // calculate energy
25 E = (M*v*v + D*x*x)/2;
26
27 // print results for debugging
28 //printf("%g %g %g %g\n", t, x, v, E);
29
30 }
31
32 // print difference from analytical solution
33 printf("%g %g %g %g\n", H, x-x_0, v-v_0, E - (M*v_0*v_0 + D*x_0*x_0)
34 /2);
35 }
36 // exit ok

```

```

37     return(0);
38 }

```

**Listing 3:** Simulation für die a) mit dem Runge Kutta Verfahren 2. Ordnung für verschiedene Schrittweiten

```

1  /*
2  Solution for a) with the Runge Kutta method (second order)
3  */
4
5  #include <stdio.h>
6  #include <math.h>
7
8  int main(){
9      // define constants
10     const double v_0 = 1., x_0=0., D=1, M=1.;
11     const double T = 1000. * M_PI;
12
13     // loop over step size H
14     for(double H = 0.001; H < 0.5; H+=0.005){
15         // initialize variables
16         double v = v_0, x = x_0;
17         double k_1_x, k_1_v, k_2_x, k_2_v, E;
18
19         // main loop
20         for(double t = 0; t < T; t += H){
21             // Runge Kutta 1. intermediate step
22             k_1_v = -H*D*x/M;
23             k_1_x = H*v;
24             // Runge Kutta 2. intermediate step
25             k_2_v = -H*D*(x + k_1_x/2)/M;
26             k_2_x = H*(v + k_1_v/2);
27             // Runge Kutta main step
28             v = v + k_2_v;
29             x = x + k_2_x;
30
31             // calculate energy
32             E = (M*v*v + D*x*x)/2;
33
34             // print results
35             //printf("%g %g %g %g\n", t, x, v, E);
36         }
37
38         // print difference from analytical solution
39         printf("%g %g %g %g\n", H, x-x_0, v-v_0, E - (M*v_0*v_0 + D*x_0*x_0))

```

```

        /2);
40    }
41
42    // exit ok
43    return(0);
44 }

```

**Listing 4:** Simulation für die a) mit dem Verlet Verfahren für verschiedene Schrittweiten

```

1  /*
2  Solution for a) with the Verlet method (second order)
3  */
4
5  #include <stdio.h>
6  #include <math.h>
7
8  int main(){
9      // define constants
10     const double v_0 = 1., x_0=0., D=1, M=1.;
11     const double T = 1000. * M_PI;
12
13     // loop over step size H
14     for(double H = 0.001; H < 0.5; H+=0.001){
15         // initialize variables
16         double v = v_0, xalt = x_0, x=xalt + H*v, xneu, E;
17
18         // main loop
19         for(double t = 0; t < T; t += H){
20             // Verlet step
21             v = (x-xalt)/H;
22             xneu = 2*x - H*H*(D/M*x) - xalt;
23             xalt = x;
24             x = xneu;
25
26             // calculate energy
27             E = (M*v*v + D*x*x)/2;
28
29             // print results
30             //printf("%g %g %g %g\n", t, x, v, E);
31
32         }
33
34         // print difference from analytical solution
35         printf("%g %g %g %g\n", H, x-x_0, v-v_0, E - (M*v_0*v_0 + D*x_0*x_0)
/2);

```



```
36     }  
37  
38     // exit ok  
39     return(0);  
40 }
```

b)

**Listing 5:** Simulation für die b) mit dem Verlet Verfahren und Dämpfung. Für verschiedene Dämpfungen wurde neu Kompiliert.

```

1  /*
2  Solution for b) with the Verlet method and damping
3  */
4
5  #include <stdio.h>
6
7  // damping constant (changed for different damping factors)
8  #define GAMMA 0.3
9
10 int main(){
11     // define constants
12     const double v_0 = 1., x_0=0., D=1, M=1.;
13     const double T = 30.;
14     const double H = 0.01;
15
16     // initialize variables
17     double v = v_0, xalt = x_0, x=xalt + H*v, xneu, E;
18
19     // main loop
20     for(double t = 0; t < T; t += H){
21         // Verlet step
22         v = (x-xalt)/H;
23         xneu = 2*x - H*H*(D/M*x + GAMMA/M * v) - xalt;
24         xalt = x;
25         x = xneu;
26
27         // calculate energy
28         E = (M*v*v + D*x*x)/2;
29
30         // print results
31         printf("%g %g %g %g\n", t, x, v, E);
32
33     }
34
35     // exit ok
36     return(0);
37 }

```

**Listing 6:** Simulation für die c,d,e) mit dem Verlet Verfahren für verschiedene Dämpfungen und Anregungsfrequenzen

```

1  /*
2     solution for part c with the Verlet method
3     includes damping and excitation
4  */
5
6  #include <math.h>
7  #include <stdio.h>
8
9  int main(){
10     // define constants
11     const double v_0 = 1., x_0=0., D=1, M=1.;
12     const double T = 300., einschwingzeit = T*0.6;
13     const double H = 0.003;
14
15     // and initialize variables
16     double v = v_0, xalt = x_0, x=xalt + H*v, xneu, E;
17     double amplitude, phase, phase2;
18
19     // loop over excitation frequencies with linear step size
20     for(double omega_t=0.001; omega_t < 3; omega_t += 0.01+0.005*omega_t)
21     {
22         // loop over different damping factors
23         for(double gamma=0; gamma < 10; gamma += 0.05){
24             // reset variables
25             amplitude = 0.0;
26             phase = 0.0;
27             phase2=0;
28             // and initial conditions
29             v = v_0;
30             xalt = x_0;
31             x=xalt + H*v;
32
33             // main simulation time loop
34             for(double t = 0; t < T; t += H){
35                 // verlet step
36                 v = (x-xalt)/H;
37                 xneu = 2*x - H*H*(D/M*x + gamma/M * v + cos(omega_t * t))
38                 - xalt;
39                 xalt = x;

```

---

```
39         x = xneu;
40
41         // print results (for debugging)
42         //printf("%g %g %g %g\n", t, x, v, E);
43
44         // test for maximum after einschwingzeit
45         if(t > einschwingzeit && x > amplitude){
46             amplitude = x;
47             phase = ((omega_t*t)/(2*M_PI)-(int)((omega_t*t)/(2*M_PI)))*2*
M_PI;
48             //phase=fmod(t*omega_t, 2*M_PI);
49
50         }
51
52     }
53
54     // print results
55     printf("%g %g %g %g\n", gamma, omega_t, amplitude, phase);
56
57 }
58
59 // print newline after each damping factor
60 printf("\n");
61 }
62
63 // return ok
64 return(0);
65 }
```