Computerphysik I: Blatt 03

Aurel Müller-Schönau und Leon Oleschko

20. Mai 2022

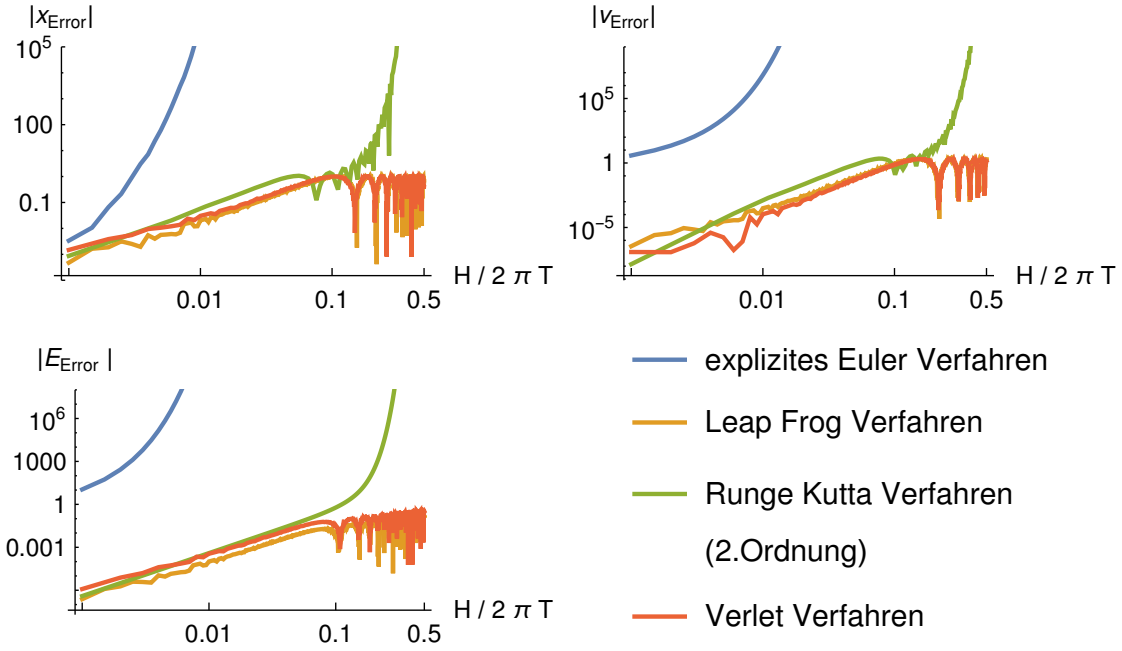

Abbildung 1: Fehler in Abhängigkeit von der Schrittweite H

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 Zeitauflösungen 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) Phasendiagram

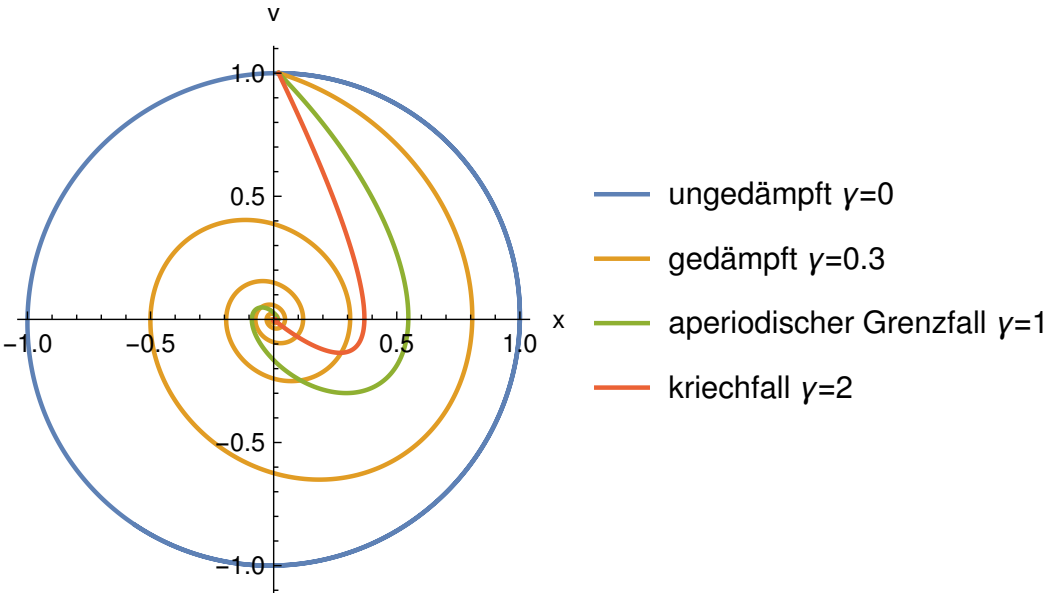

Abbildung 2: Phasendiagram für verschiedene Dämpfungen γ

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

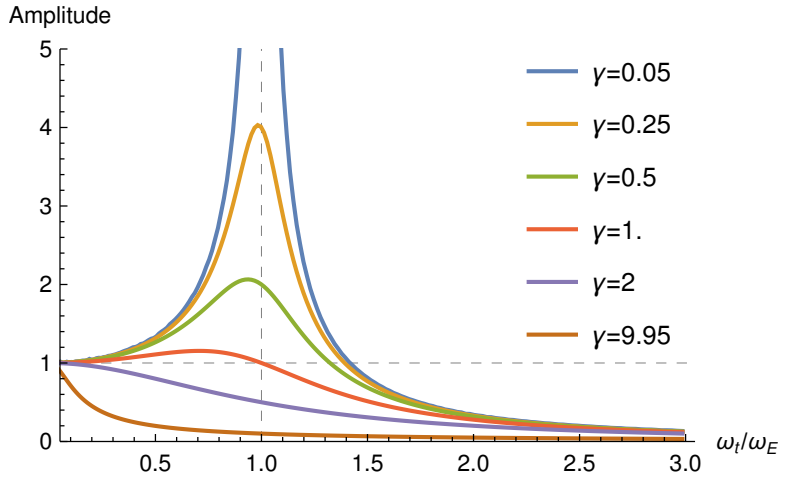

Abbildung 3: Resonanzkurve für 6 verschiedene Dämpfungen γ

c,d,e) Angetriebener Oszillator

c) In der Abbildung 3 ist die Resonanzkurve eines harmonischen Oszillators mit Eingenfrequenz ω_E, mit einer Antreibenden Frequenz ω_t und einer Amplitude der Antreibenden Kraft von 1 für 6 verschiedene Dämpfungen γ 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 $\omega_T \cdot t \mod 2\pi$ 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 π bei der Resonanzkatastrophe bei schwacher Dämpfung zu sehen.

d) In Abbildung 5 ist das Phasendiagramme für verschiedene Dämpfungen γ als 3 dimensionale Oberfläche dargestellt.

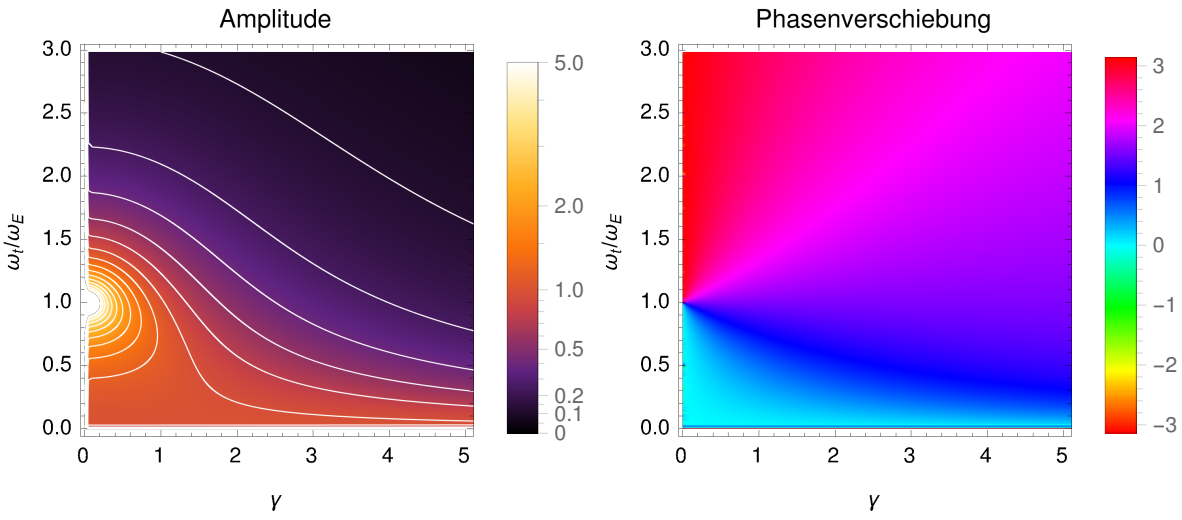

Abbildung 4: 2D Resonanzkurve für verschiedene Dämpfungen γ mit Phasenverschiebung zur antreibenden Schwingung in rad.

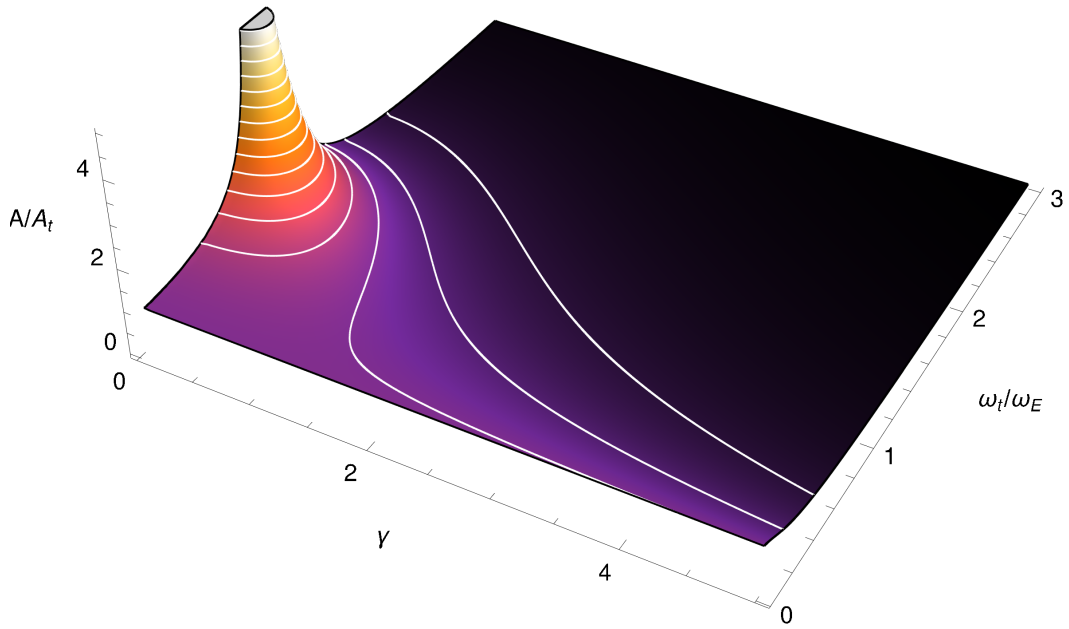

Abbildung 5: 3D Darstellung der Resonanzkurve für verschiedene Dämpfungen γ

Appendix - Code

a)

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

```
/*
Solution for a) with the explicit euler method
*/

#include <stdio.h>
#include <math.h>

int main(){
  // define constants
  const double v_0 = 1., x_0=0., D=1, M=1.;
  const double T = 1000. * M_PI;

  // loop over step size H
  for(double H = 0.001; H < 0.5; H+=0.0005){
    // initialize variables
    double v = v_0, x = x_0, xneu, vneu, E;

    // simulation loop
    for(double t = 0; t < T; t += H){
      // euler step
      vneu = v - H*D*x/M;
      xneu = x + H*v;

      // update variables
      x = xneu;
      v = vneu;

      // calculate energy
      E = (M*v*v + D*x*x)/2;

      // print results for debugging
      //printf("%g %g %g %g\n", t, x, v, E);

    }

    // print difference from analytical solution
    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);
  }
```

```
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)
       /2);
34   }
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);
  }

  // exit ok
  return(0);
}
```

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

```
/*
Solution for a) with the Verlet method (second order)
*/

#include <stdio.h>
#include <math.h>

int main(){
  // define constants
  const double v_0 = 1., x_0=0., D=1, M=1.;
  const double T = 1000. * M_PI;

  // loop over step size H
  for(double H = 0.001; H < 0.5; H+=0.001){
    // initialize variables
    double v = v_0, xalt = x_0, x=xalt + H*v, xneu, E;

    // main loop
    for(double t = 0; t < T; t += H){
      // Verlet step
      v = (x-xalt)/H;
      xneu = 2*x - H*H*(D/M*x) - xalt;
      xalt = x;
      x = xneu;

      // calculate energy
      E = (M*v*v + D*x*x)/2;

      // print results
      //printf("%g %g %g %g\n", t, x, v, E);

    }

    // print difference from analytical solution
    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 Kompaliert.

```
/*
  Solution for b) with the Verlet method and damping
*/

#include <stdio.h>

// damping constant (changed for different damping factors)
#define GAMMA 0.3

int main(){
  // define constants
  const double v_0 = 1., x_0=0., D=1, M=1.;
  const double T = 30.;
  const double H = 0.01;

  // initialize variables
  double v = v_0, xalt = x_0, x=xalt + H*v, xneu, E;

  // main loop
  for(double t = 0; t < T; t += H){
    // Verlet step
    v = (x-xalt)/H;
    xneu = 2*x - H*H*(D/M*x + GAMMA/M * v) - xalt;
    xalt = x;
    x = xneu;

    // calculate energy
    E = (M*v*v + D*x*x)/2;

    // print results
    printf("%g %g %g %g\n", t, x, v, E);

  }

  // exit ok
  return(0);
}
```

c,d,e)

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

```
/*
  solution for part c with the Verlet method
  includes damping and excitation
*/

#include <math.h>
#include <stdio.h>

int main(){
  // define constants
  const double v_0 = 1., x_0=0., D=1, M=1.;
  const double T = 300., einschwingzeit = T*0.6;
  const double H = 0.003;

  // and initialize variables
  double v = v_0, xalt = x_0, x=xalt + H*v, xneu, E;
    double amplitude, phase,phase2;

  // loop over excitation frequencies with linear step size
    for(double omega_t=0.001; omega_t < 3; omega_t += 0.01+0.005*omega_t)
    {

    // loop over different damping factors
        for(double gamma=0; gamma < 10; gamma += 0.05){
      // reset variables
            amplitude = 0.0;
            phase = 0.0;
            phase2=0;
            // and initial conditions
            v = v_0;
            xalt = x_0;
            x=xalt + H*v;

      // main simulation time loop
            for(double t = 0; t < T; t += H){
                // verlet step
                v = (x-xalt)/H;
                xneu = 2*x - H*H*(D/M*x + gamma/M * v + cos(omega_t * t))
     - xalt;
                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  }
```