Euler-Integration

ℹ Lernziele

Die Studierenden sollen…

- … das explizite Euler-Verfahren als einfachstes numerisches Integrationsverfahren beschreiben können.
- … die Herleitung des Euler-Verfahrens aus der Annahme konstanter Steigung erklären können.
- … die Energiedrift des Euler-Verfahrens im Phasenraum erklären können.
- … das Euler-Verfahren auf skalare und vektorwertige Differentialgleichungen anwenden können.
- … das Leapfrog-Verfahren zur verbesserten Energieerhaltung anwenden können.

Einführung

Nur ein kleiner Teil von Differentialgleichungen lässt sich analytisch lösen. In den meisten praktischen Anwendungen muss daher ein numerischer Zugang gewählt werden. Das einfachste und zugleich grundlegendste numerische Integrationsverfahren für gewöhnliche Differentialgleichungen ist das explizite Euler-Verfahren, das auf Leonhard Euler zurückgeht.

Die zentrale Idee des Euler-Verfahrens beruht auf einer einfachen geometrischen Überlegung. Wenn man die Lösung einer Differentialgleichung an einem bestimmten Punkt kennt, dann gibt die Differentialgleichung selbst die Steigung der Lösungskurve an diesem Punkt an. Man kann daher die Lösung näherungsweise entlang der Tangente fortsetzen und erhält so einen Schätzwert für die Lösung an einem benachbarten Punkt.

Dieses Kapitel entwickelt das Euler-Verfahren zunächst anhand eines einfachen skalaren Problems und wendet es dann auf ein vektorwertiges System an. Die Analyse des Phasenraums offenbart dabei eine fundamentale Schwäche des Verfahrens, die durch das Leapfrog-Verfahren behoben werden kann.

Heuristische Herleitung des Euler-Verfahrens

Betrachten wir ein Anfangswertproblem der Form

$$\frac{\mathrm{d}y}{\mathrm{d}t} = g(y,t), \quad y(t_0) = y_0. \tag{1}$$

Die Differentialgleichung gibt an jedem Punkt (t, y) die Steigung der Lösungskurve an. Wenn wir die Lösung zum Zeitpunkt t_i kennen, also $y_i = y(t_i)$, dann ist die Steigung der Lösungskurve durch $g(y_i, t_i)$ gegeben.

Die grundlegende Annahme des Euler-Verfahrens besteht darin, dass die Steigung über einen kleinen Zeitschritt Δt näherungsweise konstant bleibt. Unter dieser Annahme lässt sich die Lösung zum nächsten Zeitpunkt $t_{i+1} = t_i + \Delta t$ durch lineare Extrapolation berechnen:

$$y_{i+1} = y_i + \Delta t \cdot g(y_i, t_i). \tag{2}$$

Diese Formel bildet das explizite Euler-Verfahren. Der Name "explizit" bezieht sich darauf, dass y_{i+1} direkt aus den bekannten Größen y_i und t_i berechnet werden kann, ohne dass eine implizite Gleichung gelöst werden muss.

Die geometrische Interpretation ist besonders anschaulich: Man bewegt sich vom aktuellen Punkt (t_i, y_i) entlang der Tangente an die Lösungskurve um einen Schritt Δt in t-Richtung und $\Delta t \cdot g(y_i, t_i)$ in y-Richtung. Da die wahre Lösungskurve im Allgemeinen gekrümmt ist, weicht die Tangente von der tatsächlichen Lösung ab, und es entsteht ein Fehler.

ℹ Das implizite Euler-Verfahren

Eine alternative Formulierung ergibt sich, wenn man statt der Steigung am Anfang die Steigung am Ende des Intervalls verwendet:

$$y_{i+1} = y_i + \Delta t \cdot g(y_{i+1}, t_{i+1}).$$

Diese Vorschrift heißt *implizites Euler-Verfahren* oder *Rückwärts-Euler-Verfahren*. Der entscheidende Unterschied besteht darin, dass die rechte Seite von der noch unbekannten Größe y_{i+1} abhängt. Um y_{i+1} zu bestimmen, muss daher in jedem Zeitschritt eine (im Allgemeinen nichtlineare) Gleichung gelöst werden.
Dieser zusätzliche Aufwand wird durch verbesserte Stabilitätseigenschaften kompensiert. Das implizite Euler-Verfahren ist *unbedingt stabil*, das heißt es bleibt auch bei großen Zeitschritten stabil, während das explizite Verfahren für $\Delta t > 2\tau$ bei der Relaxationsgleichung zu oszillieren beginnt. Für sogenannte *steife* Differentialgleichungen, die sehr unterschiedliche Zeitskalen enthalten, sind implizite Verfahren daher oft die einzige praktikable Wahl.

Anwendung auf ein skalares Problem

Als erstes Beispiel betrachten wir die Relaxationsgleichung, die in vielen physikalischen Kontexten auftritt. Sie beschreibt den exponentiellen Zerfall einer Größe y mit der charakteristischen Zeitkonstante τ:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\frac{y}{\tau}, \quad y(0) = y_0. \tag{3}$$

Die analytische Lösung dieses Anfangswertproblems ist bekannt:

$$y(t) = y_0 \exp\left(-\frac{t}{\tau}\right). \tag{4}$$

Diese exakte Lösung dient als Referenz, um die Genauigkeit des numerischen Verfahrens zu beurteilen.

Die Anwendung des Euler-Verfahrens auf die Relaxationsgleichung führt zu der Rekursionsformel

$$y_{i+1} = y_i - \frac{\Delta t}{\tau} y_i = \left(1 - \frac{\Delta t}{\tau}\right) y_i. \tag{5}$$

Diese Formel zeigt, dass der Wert y_i in jedem Zeitschritt mit dem Faktor $(1-\Delta t/\tau)$ multipliziert wird. Nach n Schritten ergibt sich daher

$$y_n = \left(1 - \frac{\Delta t}{\tau}\right)^n y_0. \tag{6}$$

Für kleine Zeitschritte $\Delta t \ll \tau$ gilt $\left(1 - \frac{\Delta t}{\tau}\right)^n \approx \exp(-n\Delta t/\tau) = \exp(-t_n/\tau)$, was der exakten Lösung entspricht. Bei größeren Zeitschritten weicht die numerische Lösung jedoch zunehmend von der analytischen ab.

```python
import numpy as np
import matplotlib.pyplot as plt

# Parameter
tau = 2.0
y0 = 10.0
t_end = 10.0

def g(y, t, tau):
    """Rechte Seite der Relaxationsgleichung"""
    return -y / tau

def euler_integration(g, y0, t0, t_end, dt, *args):
```

```
    """
    Explizites Euler-Verfahren

    Die Funktion integriert die Differentialgleichung dy/dt = g(y, t)
    vom Anfangszeitpunkt t0 bis zum Endzeitpunkt t_end mit dem
    konstanten Zeitschritt dt.
    """
    t = t0
    y = y0
    t_list = [t]
    y_list = [y]

    while t < t_end:
        dy_dt = g(y, t, *args)
        y = y + dy_dt * dt
        t = t + dt
        t_list.append(t)
        y_list.append(y)

    return np.array(t_list), np.array(y_list)

# Analytische Lösung
t_exact = np.linspace(0, t_end, 500)
y_exact = y0 * np.exp(-t_exact / tau)

# Numerische Lösungen für verschiedene Zeitschritte
dt_values = [2.0, 1.0, 0.5, 0.1]
colors = ['red', 'orange', 'green', 'blue']

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Linker Plot: Lösungsverläufe
ax1.plot(t_exact, y_exact, 'k-', linewidth=2, label='Analytisch')

for dt, color in zip(dt_values, colors):
    t_num, y_num = euler_integration(g, y0, 0, t_end, dt, tau)
    ax1.plot(t_num, y_num, 'o-', color=color, markersize=4,
             linewidth=1, label=f'$\\Delta t = {dt}$')

ax1.set_xlabel('Zeit $t$')
ax1.set_ylabel('$y(t)$')
ax1.set_title('Lösung der Relaxationsgleichung')
```

```
ax1.legend()
ax1.grid(True, alpha=0.3)
ax1.set_xlim([0, t_end])
ax1.set_ylim([0, 12])

# Rechter Plot: Relativer Fehler am Endpunkt
dt_range = np.logspace(-2, 0.5, 30)
errors = []

for dt in dt_range:
    t_num, y_num = euler_integration(g, y0, 0, t_end, dt, tau)
    # Interpoliere auf t_end
    y_final = y_num[-1]
    t_final = t_num[-1]
    y_exact_final = y0 * np.exp(-t_final / tau)
    error = np.abs(y_final - y_exact_final) / np.abs(y_exact_final)
    errors.append(error)

ax2.loglog(dt_range, errors, 'bo-', markersize=4)
ax2.loglog(dt_range, dt_range / tau, 'k--', alpha=0.5,
           label=r'$\mathcal{O}(\Delta t)$')
ax2.set_xlabel('Zeitschritt $\\Delta t$')
ax2.set_ylabel('Relativer Fehler')
ax2.set_title('Konvergenz des Euler-Verfahrens')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

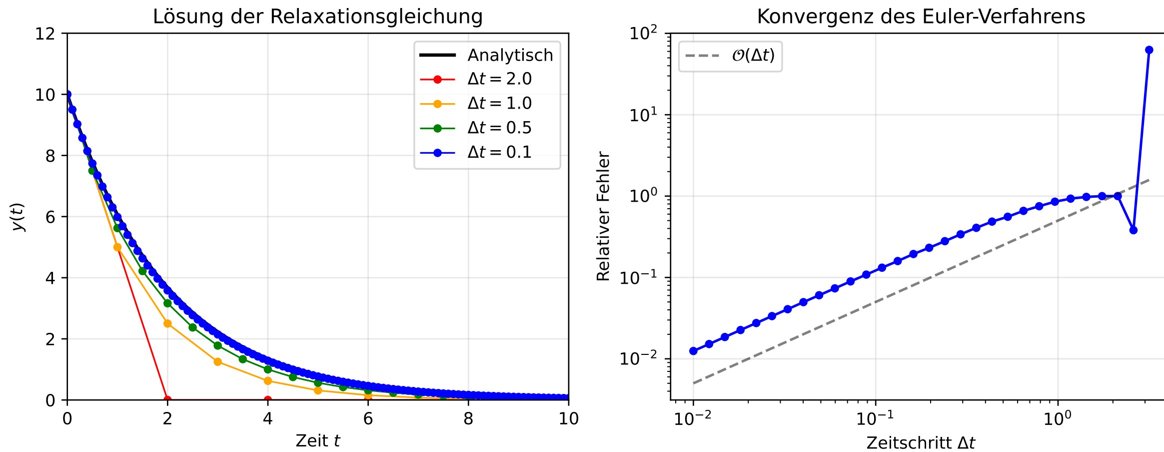

Abbildung 1: Vergleich des Euler-Verfahrens mit der analytischen Lösung der Relaxationsgleichung für verschiedene Zeitschritte. Bei kleinen Zeitschritten stimmen numerische und analytische Lösung gut überein, während bei größeren Zeitschritten deutliche Abweichungen auftreten.

Abbildung 1 demonstriert das Verhalten des Euler-Verfahrens für die Relaxationsgleichung. Der linke Teil der Abbildung zeigt, wie die numerische Lösung mit kleiner werdendem Zeitschritt immer besser mit der analytischen Lösung übereinstimmt. Der rechte Teil bestätigt, dass der Fehler linear mit dem Zeitschritt skaliert, was die erste Ordnung des Euler-Verfahrens widerspiegelt.

Vektorwertiges Problem: Das Feder-Masse-System

Die wahre Stärke numerischer Verfahren zeigt sich bei Systemen von Differentialgleichungen, die sich nicht mehr durch elementare Funktionen lösen lassen. Als prototypisches Beispiel betrachten wir das ungedämpfte Feder-Masse-System, bei dem eine Masse m an einer Feder mit Federkonstante k befestigt ist.

Die Bewegungsgleichung lautet

$$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -kx, \tag{7}$$

wobei x die Auslenkung aus der Ruhelage bezeichnet. Diese Differentialgleichung zweiter Ordnung lässt sich in ein System erster Ordnung umschreiben, indem man die Geschwindigkeit $v = \mathrm{d}x/\mathrm{d}t$ als zusätzliche Variable einführt:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v, \quad \frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{k}{m}x. \tag{8}$$

In vektorieller Schreibweise mit $\mathbf{y} = (x, v)^T$ lautet das System

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{g}(\mathbf{y}, t), \quad \text{mit} \quad \mathbf{g}(\mathbf{y}, t) = \begin{pmatrix} v \\ -\omega^2 x \end{pmatrix}, \tag{9}$$

wobei $\omega = \sqrt{k/m}$ die Eigenkreisfrequenz ist.

Das Euler-Verfahren für vektorwertige Probleme hat dieselbe Struktur wie für skalare Probleme:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \Delta t \cdot \mathbf{g}(\mathbf{y}_i, t_i). \tag{10}$$

Konkret für das Feder-Masse-System ergibt sich

$$x_{i+1} = x_i + \Delta t \cdot v_i, \quad v_{i+1} = v_i - \Delta t \cdot \omega^2 x_i. \tag{11}$$

```
import numpy as np
import matplotlib.pyplot as plt

# Parameter
k = 1.0
m = 1.0
omega = np.sqrt(k / m)

def g_spring_mass(y, t):
    """Rechte Seite des Feder-Masse-Systems"""
    x, v = y
    return np.array([v, -omega**2 * x])

def euler_vector(g, y0, t0, t_end, dt):
    """Euler-Verfahren für vektorwertige DGLs"""
    t = t0
    y = np.array(y0, dtype=float)
    t_list = [t]
    y_list = [y.copy()]

    while t < t_end - 1e-10:
        dy_dt = g(y, t)
        y = y + dy_dt * dt
        t = t + dt
        t_list.append(t)
        y_list.append(y.copy())

    return np.array(t_list), np.array(y_list)
```

```
# Anfangsbedingungen: Masse ist ausgelenkt, ruht anfangs
x0, v0 = 1.0, 0.0
t_end = 20.0

# Analytische Lösung
t_exact = np.linspace(0, t_end, 1000)
x_exact = x0 * np.cos(omega * t_exact) + v0 / omega * np.sin(omega * t_exact)
v_exact = -x0 * omega * np.sin(omega * t_exact) + v0 * np.cos(omega * t_exact)

# Verschiedene Zeitschritte
dt_values = [0.5, 0.2, 0.1]
colors = ['red', 'orange', 'green']

fig, axes = plt.subplots(2, 3, figsize=(10, 6))

for idx, (dt, color) in enumerate(zip(dt_values, colors)):
    t_num, y_num = euler_vector(g_spring_mass, [x0, v0], 0, t_end, dt)
    x_num, v_num = y_num[:, 0], y_num[:, 1]

    # Obere Reihe: Zeitverlauf
    axes[0, idx].plot(t_exact, x_exact, 'b-', linewidth=2, label='Exakt $x(t)$')
    axes[0, idx].plot(t_num, x_num, '--', color=color, linewidth=1.5,
                      label=f'Euler $x(t)$')
    axes[0, idx].set_xlabel('Zeit $t$')
    axes[0, idx].set_ylabel('Position $x$')
    axes[0, idx].set_title(f'$\\Delta t = {dt}$')
    axes[0, idx].legend(fontsize=9)
    axes[0, idx].grid(True, alpha=0.3)
    axes[0, idx].set_ylim([-3, 3])

    # Untere Reihe: Phasenraum
    axes[1, idx].plot(x_exact, v_exact, 'b-', linewidth=2, label='Exakt')
    axes[1, idx].plot(x_num, v_num, '--', color=color, linewidth=1.5,
                      label='Euler')
    axes[1, idx].set_xlabel('Position $x$')
    axes[1, idx].set_ylabel('Geschwindigkeit $v$')
    axes[1, idx].set_title('Phasenraum')
    axes[1, idx].legend(fontsize=9)
    axes[1, idx].grid(True, alpha=0.3)
    axes[1, idx].set_aspect('equal')
    axes[1, idx].set_xlim([-3, 3])
    axes[1, idx].set_ylim([-3, 3])
```

```
plt.tight_layout()
plt.show()
```

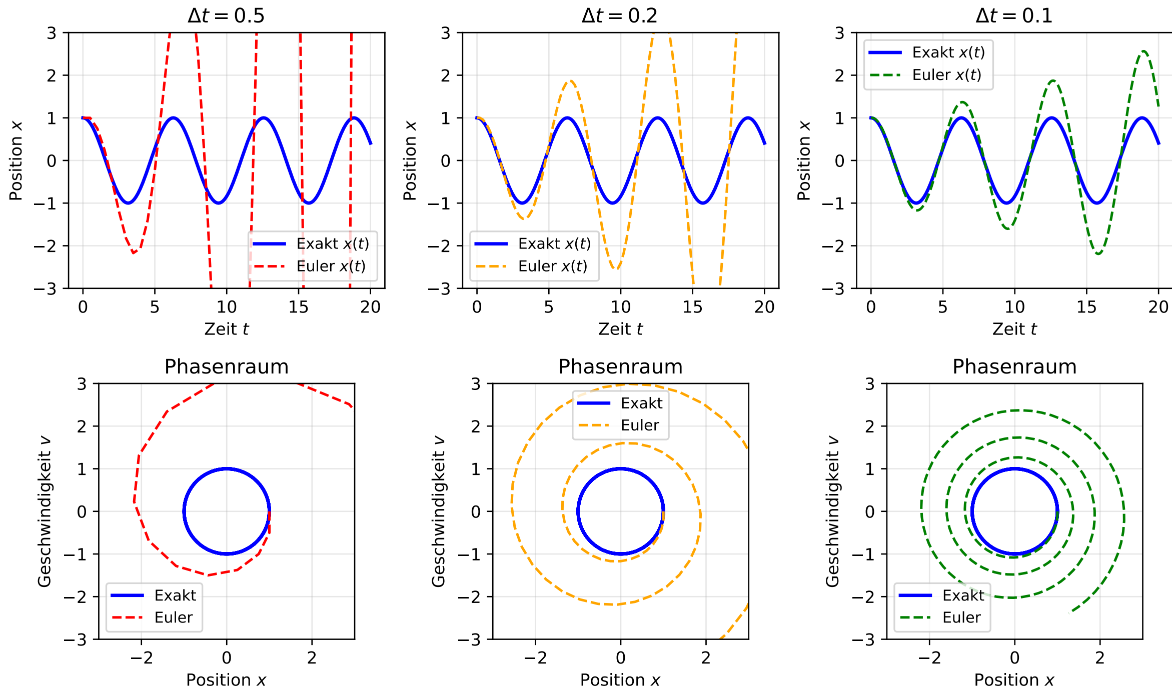

Abbildung 2: Numerische Lösung des Feder-Masse-Systems mit dem Euler-Verfahren für verschiedene Zeitschritte. Der linke Teil zeigt den Zeitverlauf von Position und Geschwindigkeit, der rechte Teil das Phasenraumportrait. Die exakte Lösung beschreibt einen Kreis im Phasenraum, während das Euler-Verfahren nach außen spiralt.

Abbildung 2 offenbart ein grundlegendes Problem des Euler-Verfahrens bei konservativen Systemen. Die exakte Lösung des ungedämpften Feder-Masse-Systems erhält die mechanische Energie $E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2$. Im Phasenraum entspricht dies einer geschlossenen Kreisbahn um den Ursprung.

Das Euler-Verfahren hingegen fügt dem System in jedem Zeitschritt künstlich Energie hinzu, was sich im Phasenraum als nach außen gerichtete Spirale manifestiert. Dieser Effekt wird als numerische Energiedrift bezeichnet und ist umso ausgeprägter, je größer der Zeitschritt gewählt wird. Selbst bei kleinen Zeitschritten akkumuliert sich die Energiedrift über lange Integrationszeiten und führt schließlich zu physikalisch unsinnigen Ergebnissen.

Die Energiedrift des Euler-Verfahrens

Um die Energiedrift des Euler-Verfahrens quantitativ zu verstehen, betrachten wir die Entwicklung der Energie über einen Zeitschritt. Die totale Energie des Feder-Masse-Systems ist

$$E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2. \tag{12}$$

Nach einem Euler-Schritt mit den Rekursionsformeln aus Gleichung 11 ergibt sich die neue Energie

$$E_{i+1} = \frac{1}{2}mv_{i+1}^2 + \frac{1}{2}kx_{i+1}^2. \tag{13}$$

Einsetzen der Euler-Formeln und Ausmultiplizieren liefert nach einiger Rechnung

$$E_{i+1} = E_i + \frac{1}{2}k\omega^2(\Delta t)^2 \left(v_i^2 + \omega^2 x_i^2\right) = E_i \left(1 + \omega^2(\Delta t)^2\right). \tag{14}$$

Die Energie wächst also in jedem Zeitschritt um den Faktor $(1 + \omega^2 \Delta t^2)$. Nach n Schritten hat sich die Energie auf

$$E_n = E_0 \left(1 + \omega^2(\Delta t)^2\right)^n \tag{15}$$

erhöht. Für kleine Zeitschritte gilt $\left(1 + \omega^2(\Delta t)^2\right)^n \approx \exp(n\omega^2(\Delta t)^2) = \exp(\omega^2 \Delta t \cdot t)$, sodass die Energie exponentiell mit der Zeit wächst.

```
import numpy as np
import matplotlib.pyplot as plt

# Parameter
omega = 1.0
x0, v0 = 1.0, 0.0
E0 = 0.5 * omega**2 * x0**2 + 0.5 * v0**2

def euler_with_energy(omega, x0, v0, t_end, dt):
    """Euler-Integration mit Energieberechnung"""
    t, x, v = 0, x0, v0
    t_list, E_list = [t], [0.5 * omega**2 * x**2 + 0.5 * v**2]

    while t < t_end:
        x_new = x + dt * v
        v_new = v - dt * omega**2 * x
        x, v = x_new, v_new
        t += dt
        E = 0.5 * omega**2 * x**2 + 0.5 * v**2
```

```
        t_list.append(t)
        E_list.append(E)

    return np.array(t_list), np.array(E_list)

t_end = 50.0
dt_values = [0.3, 0.2, 0.1, 0.05]
colors = ['red', 'orange', 'green', 'blue']

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

for dt, color in zip(dt_values, colors):
    t_num, E_num = euler_with_energy(omega, x0, v0, t_end, dt)
    ax1.plot(t_num, E_num / E0, color=color, linewidth=1.5,
             label=f'$\\Delta t = {dt}$')

ax1.axhline(y=1, color='k', linestyle='--', alpha=0.5, label='Exakt')
ax1.set_xlabel('Zeit $t$')
ax1.set_ylabel('$E(t) / E_0$')
ax1.set_title('Relative Energie über der Zeit')
ax1.legend()
ax1.grid(True, alpha=0.3)
ax1.set_ylim([0.9, 5])

# Wachstumsrate der Energie
dt_range = np.logspace(-2, -0.3, 20)
growth_rates = []

for dt in dt_range:
    t_num, E_num = euler_with_energy(omega, x0, v0, 20.0, dt)
    # Bestimme Wachstumsrate durch Fit
    log_E = np.log(E_num / E0)
    rate = (log_E[-1] - log_E[0]) / t_num[-1]
    growth_rates.append(rate)

ax2.loglog(dt_range, growth_rates, 'bo-', markersize=4)
ax2.loglog(dt_range, omega**2 * dt_range, 'k--', alpha=0.5,
           label=r'$\omega^2 \Delta t$')
ax2.set_xlabel('Zeitschritt $\\Delta t$')
ax2.set_ylabel('Wachstumsrate $\\gamma$')
ax2.set_title('Energiewachstumsrate')
ax2.legend()
```

```
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

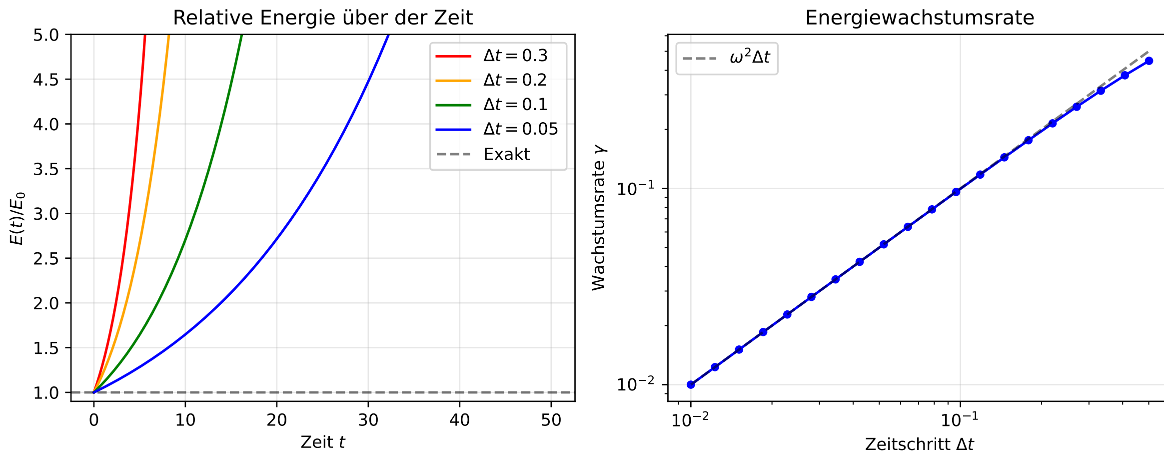

Abbildung 3: Energiedrift des Euler-Verfahrens für das Feder-Masse-System. Die Energie wächst exponentiell mit der Zeit, wobei die Wachstumsrate quadratisch vom Zeitschritt abhängt.

Abbildung 3 bestätigt die theoretische Analyse. Die linke Abbildung zeigt das exponentielle Wachstum der Energie für verschiedene Zeitschritte, während die rechte Abbildung demonstriert, dass die Wachstumsrate linear mit dem Zeitschritt skaliert. Diese Energiedrift ist ein fundamentales Problem des Euler-Verfahrens bei der Integration Hamiltonscher Systeme und motiviert die Entwicklung energieerhaltender Verfahren.

Das Leapfrog-Verfahren

Das Leapfrog-Verfahren, auch Störmer-Verlet-Verfahren genannt, ist ein symplektisches Integrationsverfahren, das die Energieerhaltung wesentlich besser approximiert als das Euler-Verfahren. Es gehört zur Klasse der Prädiktor-Korrektor-Verfahren und lässt sich als symmetrische Kombination zweier halber Euler-Schritte verstehen.

Prädiktor-Korrektor-Struktur

Das Leapfrog-Verfahren besteht aus drei Teilschritten, die sich als Prädiktor- und Korrektorschritte interpretieren lassen. Im ersten Schritt, dem **Prädiktor**, wird die Geschwindigkeit

um einen halben Zeitschritt vorwärts propagiert, wobei nur die bekannte Position x_i verwendet wird:

$$v_{i+1/2} = v_i - \frac{\Delta t}{2}\omega^2 x_i. \tag{16}$$

Diese Zwischengeschwindigkeit $v_{i+1/2}$ stellt eine Vorhersage für die Geschwindigkeit in der Mitte des Zeitintervalls dar.

Im zweiten Schritt wird die Position um einen vollen Zeitschritt propagiert, wobei die soeben berechnete Zwischengeschwindigkeit verwendet wird:

$$x_{i+1} = x_i + \Delta t \cdot v_{i+1/2}. \tag{17}$$

Dieser Schritt nutzt die verbesserte Geschwindigkeitsschätzung in der Intervallmitte und erreicht dadurch eine höhere Genauigkeit als ein einfacher Euler-Schritt.

Im dritten Schritt, dem **Korrektor**, wird die Geschwindigkeit auf den vollen Zeitschritt vervollständigt. Dabei wird nun die neue Position x_{i+1} verwendet:

$$v_{i+1} = v_{i+1/2} - \frac{\Delta t}{2}\omega^2 x_{i+1}. \tag{18}$$

Der Korrektor verwendet also Information vom Ende des Intervalls, um die Geschwindigkeitsberechnung zu verbessern.

Kompakte Darstellung

Die drei Teilschritte lassen sich auch kompakter schreiben, indem man die Zwischengeschwindigkeit $v_{i+1/2}$ eliminiert. Durch Addition von Gleichung 16 und Gleichung 18 erhält man:

$$v_{i+1} = v_i - \frac{\Delta t}{2}\omega^2 (x_i + x_{i+1}). \tag{19}$$

Diese Darstellung zeigt, dass das Leapfrog-Verfahren den Mittelwert der Beschleunigungen am Anfang und Ende des Intervalls verwendet, was der Trapezregel für numerische Integration entspricht. Zusammen mit Gleichung 17 ergibt sich das vollständige Leapfrog-Schema.

Symplektizität

Die besondere Eigenschaft des Leapfrog-Verfahrens ist seine Symplektizität. Ein symplektisches Verfahren erhält das Phasenraumvolumen exakt und approximiert die Energie zwar nicht exakt, aber oszilliert um den wahren Wert, anstatt systematisch davon abzuweichen. Diese Eigenschaft folgt direkt aus der symmetrischen Prädiktor-Korrektor-Struktur: Der halbe Geschwindigkeitsschritt zu Beginn und der halbe Geschwindigkeitsschritt am Ende sind zueinander adjungiert, was die Zeitumkehrsymmetrie des Verfahrens garantiert. Für Langzeitintegrationen ist dies ein entscheidender Vorteil.

Beweis der Symplektizität

Wir können die Symplektizität des Leapfrog-Verfahrens direkt nachweisen. Das Verfahren lässt sich in Matrixform schreiben:

$$\begin{pmatrix} x_{i+1} \\ v_{i+1} \end{pmatrix} = \mathbf{M} \begin{pmatrix} x_i \\ v_i \end{pmatrix}.$$

Mit $\lambda = (\omega \Delta t)^2$ ergibt sich aus den drei Teilschritten (Gleichung 16, Gleichung 17, Gleichung 18) die Matrix:

$$\mathbf{M} = \begin{pmatrix} 1 - \frac{\lambda}{2} & \Delta t \\ -\omega^2 \Delta t \left(1 - \frac{\lambda}{4}\right) & 1 - \frac{\lambda}{2} \end{pmatrix}.$$

Die Determinante dieser Matrix berechnet sich zu:

$$\det(\mathbf{M}) = \left(1 - \frac{\lambda}{2}\right)^2 + \lambda \left(1 - \frac{\lambda}{4}\right) = 1 - \lambda + \frac{\lambda^2}{4} + \lambda - \frac{\lambda^2}{4} = 1.$$

Da $\det(\mathbf{M}) = 1$, ist die Transformation flächenerhaltend im Phasenraum (x, v). In zwei Dimensionen ist Flächenerhaltung äquivalent zur Symplektizität, was die Langzeitstabilität des Verfahrens erklärt.

```python
import numpy as np
import matplotlib.pyplot as plt

def leapfrog(omega, x0, v0, t_end, dt):
    """Leapfrog-Verfahren für den harmonischen Oszillator"""
    t = 0
    x, v = x0, v0
    t_list = [t]
    x_list = [x]
    v_list = [v]

    while t < t_end - 1e-10:
        # Halber Geschwindigkeitsschritt
        v_half = v - 0.5 * dt * omega**2 * x
        # Voller Positionsschritt
        x = x + dt * v_half
        # Halber Geschwindigkeitsschritt
        v = v_half - 0.5 * dt * omega**2 * x
        t += dt

        t_list.append(t)
```

```
        x_list.append(x)
        v_list.append(v)

    return np.array(t_list), np.array(x_list), np.array(v_list)

def euler_oscillator(omega, x0, v0, t_end, dt):
    """Euler-Verfahren für den harmonischen Oszillator"""
    t = 0
    x, v = x0, v0
    t_list = [t]
    x_list = [x]
    v_list = [v]

    while t < t_end - 1e-10:
        x_new = x + dt * v
        v_new = v - dt * omega**2 * x
        x, v = x_new, v_new
        t += dt

        t_list.append(t)
        x_list.append(x)
        v_list.append(v)

    return np.array(t_list), np.array(x_list), np.array(v_list)

# Parameter
omega = 1.0
x0, v0 = 1.0, 0.0
t_end = 100.0
dt = 0.2

# Analytische Lösung
t_exact = np.linspace(0, t_end, 2000)
x_exact = x0 * np.cos(omega * t_exact)
v_exact = -x0 * omega * np.sin(omega * t_exact)

# Numerische Lösungen
t_euler, x_euler, v_euler = euler_oscillator(omega, x0, v0, t_end, dt)
t_leap, x_leap, v_leap = leapfrog(omega, x0, v0, t_end, dt)

# Energie berechnen
E0 = 0.5 * omega**2 * x0**2 + 0.5 * v0**2
```

```
E_euler = 0.5 * omega**2 * x_euler**2 + 0.5 * v_euler**2
E_leap = 0.5 * omega**2 * x_leap**2 + 0.5 * v_leap**2

fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# Phasenraum Euler
axes[0, 0].plot(x_exact, v_exact, 'b-', linewidth=1, alpha=0.5, label='Exakt')
axes[0, 0].plot(x_euler, v_euler, 'r-', linewidth=1, label='Euler')
axes[0, 0].set_xlabel('Position $x$')
axes[0, 0].set_ylabel('Geschwindigkeit $v$')
axes[0, 0].set_title(f'Euler-Verfahren, $\\Delta t = {dt}$')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].set_aspect('equal')
axes[0, 0].set_xlim([-8, 8])
axes[0, 0].set_ylim([-8, 8])

# Phasenraum Leapfrog
axes[0, 1].plot(x_exact, v_exact, 'b-', linewidth=1, alpha=0.5, label='Exakt')
axes[0, 1].plot(x_leap, v_leap, 'g-', linewidth=1, label='Leapfrog')
axes[0, 1].set_xlabel('Position $x$')
axes[0, 1].set_ylabel('Geschwindigkeit $v$')
axes[0, 1].set_title(f'Leapfrog-Verfahren, $\\Delta t = {dt}$')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)
axes[0, 1].set_aspect('equal')
axes[0, 1].set_xlim([-1.5, 1.5])
axes[0, 1].set_ylim([-1.5, 1.5])

# Energie Euler
axes[1, 0].plot(t_euler, E_euler / E0, 'r-', linewidth=1, label='Euler')
axes[1, 0].axhline(y=1, color='k', linestyle='--', alpha=0.5)
axes[1, 0].set_xlabel('Zeit $t$')
axes[1, 0].set_ylabel('$E(t) / E_0$')
axes[1, 0].set_title('Energieverlauf Euler')
axes[1, 0].grid(True, alpha=0.3)
axes[1, 0].set_ylim([0.5, 100])
axes[1, 0].set_yscale('log')

# Energie Leapfrog
axes[1, 1].plot(t_leap, E_leap / E0, 'g-', linewidth=1, label='Leapfrog')
axes[1, 1].axhline(y=1, color='k', linestyle='--', alpha=0.5)
```

```python
axes[1, 1].set_xlabel('Zeit $t$')
axes[1, 1].set_ylabel('$E(t) / E_0$')
axes[1, 1].set_title('Energieverlauf Leapfrog')
axes[1, 1].grid(True, alpha=0.3)
axes[1, 1].set_ylim([0.98, 1.02])

plt.tight_layout()
plt.show()

print(f"Euler: Endenergie / Anfangsenergie = {E_euler[-1] / E0:.2f}")
print(f"Leapfrog: Endenergie / Anfangsenergie = {E_leap[-1] / E0:.6f}")
```

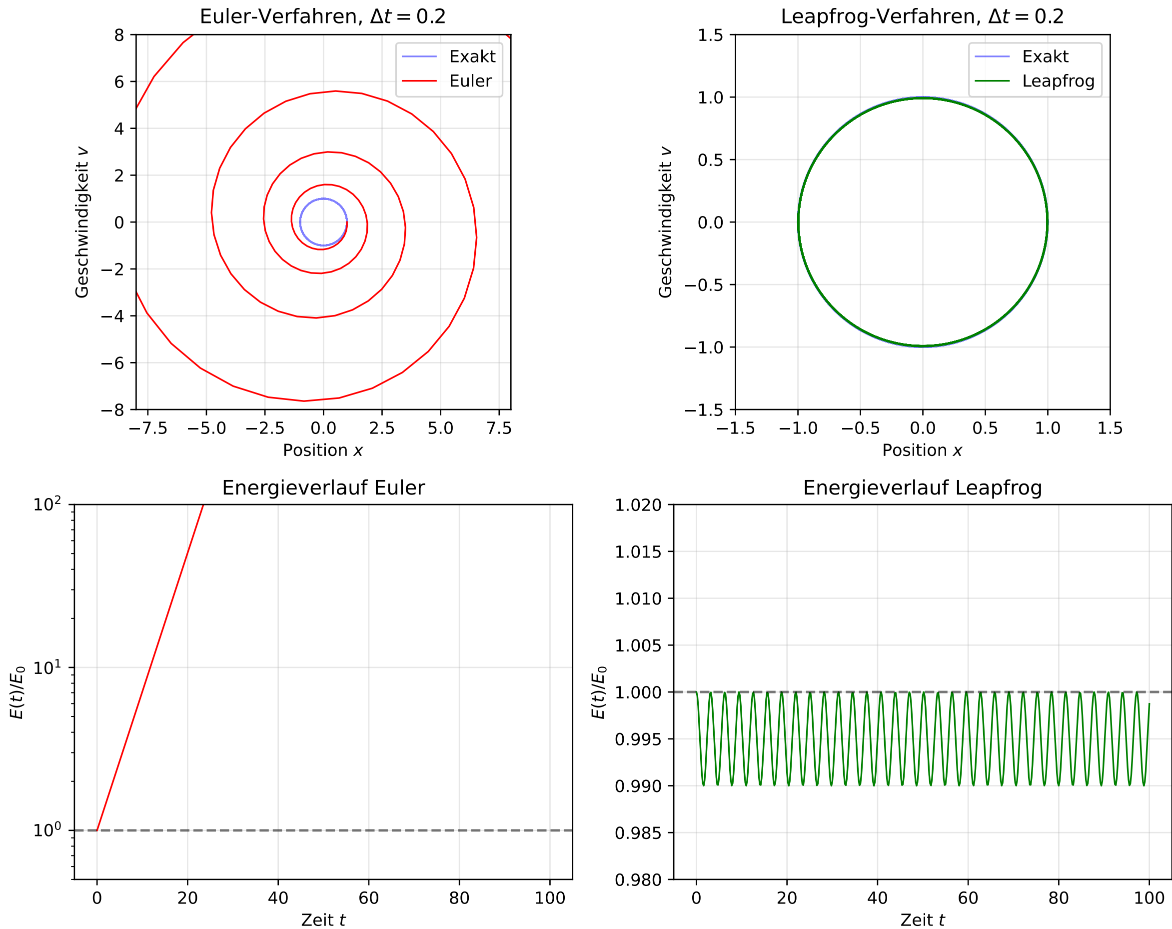

Abbildung 4: Vergleich von Euler- und Leapfrog-Verfahren für das Feder-Masse-System. Das Leapfrog-Verfahren erhält die Energie wesentlich besser und zeigt im Phasenraum eine nahezu geschlossene Kurve anstelle der nach außen spiralenden Trajektorie des Euler-Verfahrens.

```
Euler: Endenergie / Anfangsenergie = 328601581.58
Leapfrog: Endenergie / Anfangsenergie = 0.998736
```

Abbildung 4 verdeutlicht den dramatischen Unterschied zwischen Euler- und Leapfrog-Verfahren bei langen Integrationszeiten. Während das Euler-Verfahren nach $t = 100$ die Energie um mehr als das Hundertfache aufgebläht hat und entsprechend weit nach außen spiralt, bleibt das Leapfrog-Verfahren nahezu auf dem korrekten Energieniveau. Die Energie des Leapfrog-Verfahrens oszilliert zwar leicht, weicht aber im Mittel nicht systematisch vom Anfangswert ab.

Vergleich der Verfahren im Phasenraum

Die Phasenraumdarstellung bietet einen tiefen Einblick in das qualitative Verhalten numerischer Integrationsverfahren. Der Phasenraum für das Feder-Masse-System wird durch die Koordinaten (x, v) aufgespannt, wobei jeder Punkt einem bestimmten mechanischen Zustand entspricht.

Für konservative Systeme wie den ungedämpften Oszillator sind die Trajektorien im Phasenraum geschlossene Kurven, die Linien konstanter Energie entsprechen. Die Fläche innerhalb einer solchen Kurve ist proportional zur Wirkung, einer fundamentalen Größe der klassischen Mechanik. Der Satz von Liouville besagt, dass Hamiltonsche Systeme das Phasenraumvolumen erhalten.

```
import numpy as np
import matplotlib.pyplot as plt

omega = 1.0
x0, v0 = 1.0, 0.0
t_end = 40.0

# Exakte Lösung für einen Kreis im Phasenraum
theta = np.linspace(0, 2*np.pi, 200)
x_circle = x0 * np.cos(theta)
v_circle = -x0 * omega * np.sin(theta)

dt_values = [0.5, 0.3, 0.1]

fig, axes = plt.subplots(2, 3, figsize=(10, 6))

for idx, dt in enumerate(dt_values):
    # Euler
    t_e, x_e, v_e = euler_oscillator(omega, x0, v0, t_end, dt)
    # Leapfrog
    t_l, x_l, v_l = leapfrog(omega, x0, v0, t_end, dt)

    # Euler Phasenraum
    axes[0, idx].plot(x_circle, v_circle, 'b-', linewidth=2, alpha=0.5,
                      label='Exakt')
    axes[0, idx].plot(x_e, v_e, 'r-', linewidth=1, label='Euler')
    axes[0, idx].plot(x0, v0, 'ko', markersize=8)
    axes[0, idx].set_xlabel('$x$')
    axes[0, idx].set_ylabel('$v$')
    axes[0, idx].set_title(f'Euler, $\\Delta t = {dt}$')
    axes[0, idx].legend(fontsize=9)
```

```
    axes[0, idx].grid(True, alpha=0.3)
    axes[0, idx].set_aspect('equal')

    # Bestimme Achsenlimits basierend auf Euler
    max_val = max(np.max(np.abs(x_e)), np.max(np.abs(v_e)), 1.5)
    axes[0, idx].set_xlim([-max_val*1.1, max_val*1.1])
    axes[0, idx].set_ylim([-max_val*1.1, max_val*1.1])

    # Leapfrog Phasenraum
    axes[1, idx].plot(x_circle, v_circle, 'b-', linewidth=2, alpha=0.5,
                      label='Exakt')
    axes[1, idx].plot(x_l, v_l, 'g-', linewidth=1, label='Leapfrog')
    axes[1, idx].plot(x0, v0, 'ko', markersize=8)
    axes[1, idx].set_xlabel('$x$')
    axes[1, idx].set_ylabel('$v$')
    axes[1, idx].set_title(f'Leapfrog, $\\Delta t = {dt}$')
    axes[1, idx].legend(fontsize=9)
    axes[1, idx].grid(True, alpha=0.3)
    axes[1, idx].set_aspect('equal')
    axes[1, idx].set_xlim([-1.5, 1.5])
    axes[1, idx].set_ylim([-1.5, 1.5])

plt.tight_layout()
plt.show()
```

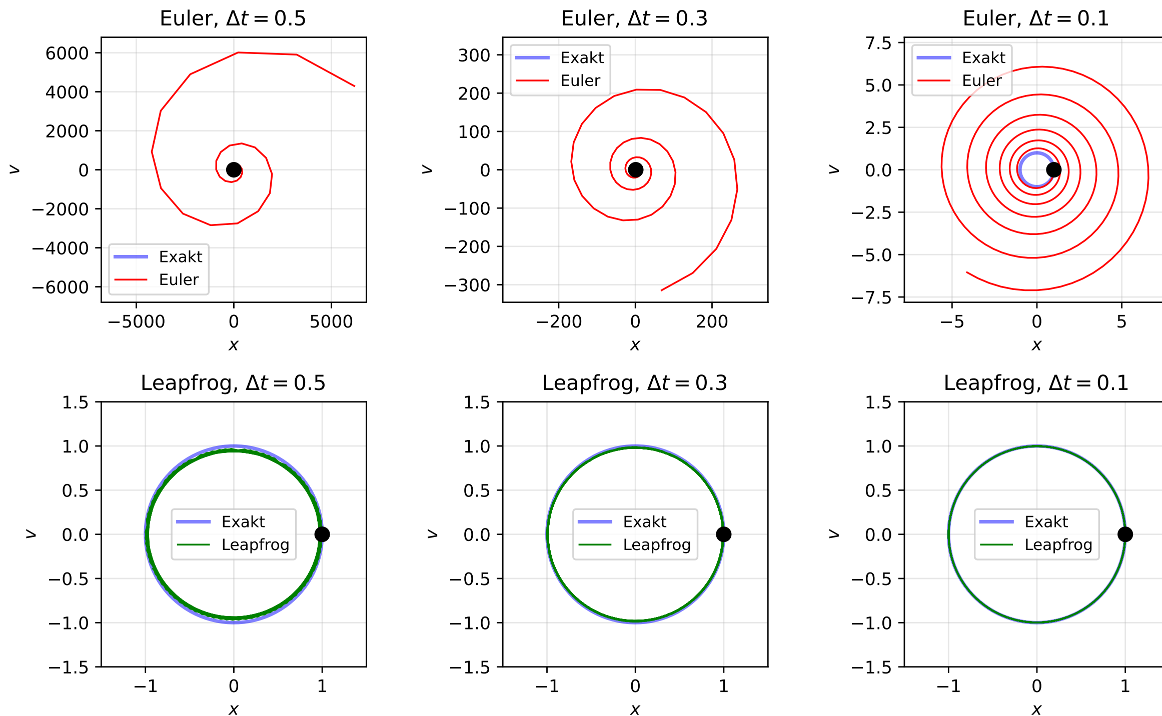

Abbildung 5: Einfluss des Zeitschritts auf die Phasenraumtrajektorien für Euler- und Leapfrog-Verfahren. Das Euler-Verfahren zeigt bei allen Zeitschritten eine nach außen gerichtete Drift, während das Leapfrog-Verfahren die geschlossene Struktur der Trajektorien bewahrt.

Abbildung 5 zeigt den systematischen Einfluss des Zeitschritts auf beide Verfahren. Das Euler-Verfahren produziert bei jedem Zeitschritt eine nach außen gerichtete Spirale, wobei die Spiralrate mit dem Zeitschritt zunimmt. Das Leapfrog-Verfahren hingegen erzeugt auch bei größeren Zeitschritten eine nahezu geschlossene Kurve, die nur leicht von der exakten Kreisbahn abweicht. Diese Abweichung manifestiert sich als geringfügige Elliptizität, aber nicht als systematische Drift.

Zusammenfassung

Das explizite Euler-Verfahren ist das einfachste numerische Integrationsverfahren für gewöhnliche Differentialgleichungen. Seine Herleitung beruht auf der Annahme, dass die Steigung der Lösungskurve über einen kleinen Zeitschritt konstant bleibt. Trotz seiner Einfachheit weist das Verfahren fundamentale Einschränkungen auf.

Für dissipative Systeme wie die Relaxationsgleichung liefert das Euler-Verfahren bei hinreichend kleinen Zeitschritten akzeptable Ergebnisse. Der Fehler skaliert linear mit dem Zeitschritt, was die erste Konvergenzordnung des Verfahrens widerspiegelt.

Bei konservativen Systemen wie dem Feder-Masse-System zeigt sich jedoch eine systematische Energiedrift. Das Euler-Verfahren fügt dem System in jedem Zeitschritt künstlich Energie hinzu, was im Phasenraum als nach außen gerichtete Spirale sichtbar wird. Für Langzeitintegrationen ist dieses Verhalten inakzeptabel.

Das Leapfrog-Verfahren bietet einen eleganten Ausweg aus diesem Dilemma. Durch die versetzte Berechnung von Position und Geschwindigkeit erhält es die symplektische Struktur des Phasenraums und vermeidet die systematische Energiedrift. Die Energie oszilliert zwar leicht um den Anfangswert, weicht aber im Mittel nicht davon ab.

> **! Important**
>
> Für die numerische Integration konservativer Systeme über lange Zeiten sollte das einfache Euler-Verfahren vermieden werden. Symplektische Verfahren wie Leapfrog oder höherwertige symplektische Runge-Kutta-Verfahren sind in solchen Fällen die bessere Wahl, da sie die fundamentalen geometrischen Eigenschaften des Phasenraums respektieren.

Im nächsten Kapitel werden wir die Runge-Kutta-Verfahren kennenlernen, die durch mehrere Funktionsauswertungen pro Zeitschritt eine höhere Genauigkeit erreichen. Diese Verfahren bilden die Grundlage moderner ODE-Löser wie `scipy.integrate.solve_ivp`.