Runge-Kutta-Verfahren

Lernziele

Die Studierenden sollen…

- … die Grundidee der Runge-Kutta-Verfahren beschreiben können.
- … den Unterschied zwischen verschiedenen Ordnungen von Runge-Kutta-Verfahren erklären können.
- … das Konzept der Konvergenzordnung erklären können.
- … Runge-Kutta-Verfahren zur numerischen Lösung von Differentialgleichungen anwenden können.
- … die Funktion `scipy.integrate.solve_ivp` verstehen und anwenden können.

Einführung

Im vorherigen Kapitel haben wir das Euler-Verfahren als einfachstes numerisches Integrationsverfahren kennengelernt. Seine Herleitung basiert auf der Annahme konstanter Steigung über einen Zeitschritt, was zu einem Verfahren erster Ordnung führt. Um eine akzeptable Genauigkeit zu erreichen, sind daher sehr kleine Zeitschritte erforderlich.

Die Runge-Kutta-Verfahren stellen eine Familie von Integrationsverfahren höherer Ordnung dar, die durch geschickte Kombination mehrerer Funktionsauswertungen pro Zeitschritt eine wesentlich bessere Genauigkeit erzielen. Diese Verfahren bilden die algorithmische Grundlage der meisten modernen Differentialgleichungslöser, einschließlich der Funktion `scipy.integrate.solve_ivp`.

Die zentrale Idee besteht darin, die Steigung der Lösungskurve nicht nur am Anfang des Intervalls auszuwerten, sondern auch an weiteren Zwischenpunkten. Durch geeignete Gewichtung dieser Steigungswerte lässt sich der Fehler systematisch reduzieren, ohne dass Ableitungen höherer Ordnung berechnet werden müssen.

Das Euler-Verfahren als Ausgangspunkt

Betrachten wir erneut das Anfangswertproblem

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{g}(\mathbf{y}, t), \quad \mathbf{y}(t_0) = \mathbf{y}_0. \tag{1}$$

Das explizite Euler-Verfahren approximiert die Lösung zum Zeitpunkt $t + \Delta t$ durch

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \Delta t\, \mathbf{g}(\mathbf{y}(t), t) + \mathcal{O}(\Delta t^2). \tag{2}$$

Der Fehlerterm $\mathcal{O}(\Delta t^2)$ bezeichnet den lokalen Abbruchfehler, also den Fehler, der in einem einzelnen Zeitschritt entsteht. Da für eine Integration über die Zeit T insgesamt $N = T/\Delta t$ Schritte benötigt werden, akkumuliert sich dieser lokale Fehler zum globalen Fehler $\mathcal{O}(\Delta t)$.

> **Konvergenzordnung**
>
> Ein numerisches Verfahren besitzt die Konvergenzordnung p, wenn der globale Fehler wie $\mathcal{O}(\Delta t^p)$ mit dem Zeitschritt skaliert. Das Euler-Verfahren hat demnach Konvergenzordnung 1. Ein Verfahren der Ordnung p hat einen lokalen Abbruchfehler von $\mathcal{O}(\Delta t^{p+1})$.

Das Heun-Verfahren

Die fundamentale Schwäche des Euler-Verfahrens liegt darin, dass es nur die Steigung am Anfang des Intervalls verwendet. Eine bessere Approximation würde die durchschnittliche Steigung über das gesamte Intervall berücksichtigen. Da die Steigung am Ende des Intervalls jedoch von der noch unbekannten Lösung $\mathbf{y}(t + \Delta t)$ abhängt, muss sie zunächst geschätzt werden.

Das Heun-Verfahren, auch als Prädiktor-Korrektor-Verfahren bekannt, realisiert diese Idee in zwei Schritten. Im ersten Schritt, dem Prädiktor, wird ein vorläufiger Schätzwert für die Lösung am Ende des Intervalls berechnet:

$$\tilde{\mathbf{y}}(t + \Delta t) = \mathbf{y}(t) + \Delta t\, \mathbf{g}(\mathbf{y}(t), t). \tag{3}$$

Im zweiten Schritt, dem Korrektor, wird die Lösung unter Verwendung des Mittelwerts der Steigungen am Anfang und am geschätzten Ende des Intervalls neu berechnet:

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \frac{\Delta t}{2} \left[\mathbf{g}(\mathbf{y}(t), t) + \mathbf{g}(\tilde{\mathbf{y}}(t + \Delta t), t + \Delta t) \right]. \tag{4}$$

Das Heun-Verfahren erreicht die Konvergenzordnung 2, was bedeutet, dass eine Halbierung des Zeitschritts den Fehler um den Faktor 4 reduziert. Im Vergleich dazu würde das Euler-Verfahren bei Halbierung des Zeitschritts den Fehler nur halbieren.

Herleitung der Konvergenzordnung

Um die Ordnung des Heun-Verfahrens zu bestimmen, vergleichen wir es mit der Taylor-Entwicklung der exakten Lösung. Für eine skalare Gleichung $y' = g(y,t)$ gilt:

$$y(t+\Delta t) = y(t) + \Delta t\, y'(t) + \frac{\Delta t^2}{2} y''(t) + \mathcal{O}(\Delta t^3).$$

Mit $y' = g$ und der Kettenregel folgt für die zweite Ableitung:

$$y'' = \frac{\mathrm{d}g}{\mathrm{d}t} = \frac{\partial g}{\partial y} y' + \frac{\partial g}{\partial t} = g\frac{\partial g}{\partial y} + \frac{\partial g}{\partial t}.$$

Damit lautet die exakte Lösung:

$$y(t+\Delta t) = y + \Delta t\, g + \frac{\Delta t^2}{2}\left(g\frac{\partial g}{\partial y} + \frac{\partial g}{\partial t}\right) + \mathcal{O}(\Delta t^3). \tag{5}$$

Für das Heun-Verfahren entwickeln wir $g(\tilde{y}, t+\Delta t)$ mit $\tilde{y} = y + \Delta t\, g$:

$$g(\tilde{y}, t+\Delta t) = g(y,t) + \Delta t\, g\frac{\partial g}{\partial y} + \Delta t\frac{\partial g}{\partial t} + \mathcal{O}(\Delta t^2).$$

Einsetzen in die Korrektor-Formel (Gleichung 4) ergibt:

$$\begin{aligned} y_{i+1} &= y + \frac{\Delta t}{2}\left[g + g + \Delta t\left(g\frac{\partial g}{\partial y} + \frac{\partial g}{\partial t}\right) + \mathcal{O}(\Delta t^2)\right] \\ &= y + \Delta t\, g + \frac{\Delta t^2}{2}\left(g\frac{\partial g}{\partial y} + \frac{\partial g}{\partial t}\right) + \mathcal{O}(\Delta t^3). \end{aligned} \tag{6}$$

Der Vergleich von Gleichung 6 mit Gleichung 5 zeigt, dass das Heun-Verfahren die exakte Lösung bis einschließlich $\mathcal{O}(\Delta t^2)$ reproduziert. Der lokale Abbruchfehler ist daher $\mathcal{O}(\Delta t^3)$, was einem Verfahren zweiter Ordnung entspricht.

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

def euler_step(g, y, t, dt):
    """Ein Schritt des Euler-Verfahrens"""
    return y + dt * g(y, t)
```

```
def heun_step(g, y, t, dt):
    """Ein Schritt des Heun-Verfahrens (RK2)"""
    k1 = g(y, t)
    y_tilde = y + dt * k1
    k2 = g(y_tilde, t + dt)
    return y + dt/2 * (k1 + k2)

def integrate(step_func, g, y0, t_span, dt):
    """Integriere eine DGL mit gegebenem Zeitschritt-Verfahren"""
    t_start, t_end = t_span
    t = np.arange(t_start, t_end + dt, dt)
    y = np.zeros((len(t), len(y0)))
    y[0] = y0

    for i in range(len(t) - 1):
        y[i+1] = step_func(g, y[i], t[i], dt)

    return t, y

# Harmonischer Oszillator
omega = 1.0
def harmonic_oscillator(y, t):
    return np.array([y[1], -omega**2 * y[0]])

# Anfangsbedingungen
y0 = np.array([1.0, 0.0])
t_span = (0, 20)

# Analytische Lösung
t_exact = np.linspace(0, 20, 500)
y_exact = 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, 5))

for idx, dt in enumerate(dt_values):
    t_e, y_e = integrate(euler_step, harmonic_oscillator, y0, t_span, dt)
    t_h, y_h = integrate(heun_step, harmonic_oscillator, y0, t_span, dt)
```

```
    # Obere Reihe: Lösungen
    axes[0, idx].plot(t_exact, y_exact, 'b-', linewidth=2, label='Exakt')
    axes[0, idx].plot(t_e, y_e[:, 0], 'r--', linewidth=1.5, label='Euler')
    axes[0, idx].plot(t_h, y_h[:, 0], 'g-.', linewidth=1.5, label='Heun')
    axes[0, idx].set_xlabel('Zeit $t$')
    axes[0, idx].set_ylabel('$y(t)$')
    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(y_exact, -omega * np.sin(omega * t_exact), 'b-',
                      linewidth=2, label='Exakt')
    axes[1, idx].plot(y_e[:, 0], y_e[:, 1], 'r--', linewidth=1.5, label='Euler')
    axes[1, idx].plot(y_h[:, 0], y_h[:, 1], 'g-.', linewidth=1.5, label='Heun')
    axes[1, idx].set_xlabel('$y$')
    axes[1, idx].set_ylabel("$y'$")
    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()
```

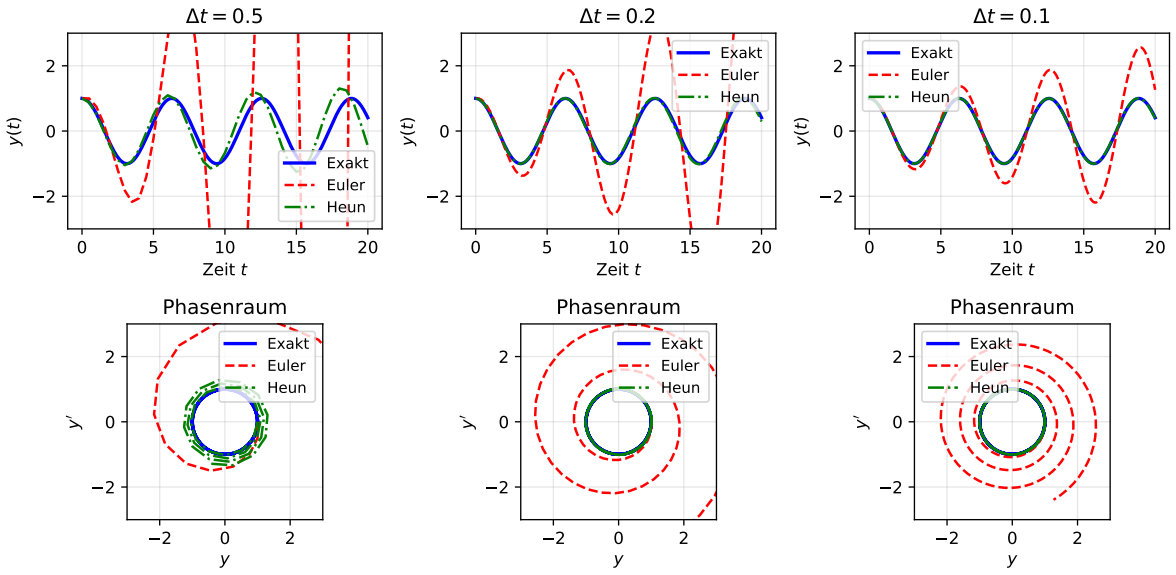

Abbildung 1: Vergleich von Euler- und Heun-Verfahren für den harmonischen Oszillator. Das Heun-Verfahren zeigt bei gleichem Zeitschritt eine deutlich bessere Energieerhaltung als das Euler-Verfahren.

Abbildung 1 verdeutlicht den qualitativen Unterschied zwischen Euler- und Heun-Verfahren am Beispiel des harmonischen Oszillators. In der oberen Reihe zeigt der Zeitverlauf der Lösung, dass das Euler-Verfahren bei größeren Zeitschritten eine wachsende Amplitude aufweist, während das Heun-Verfahren die Schwingung wesentlich besser erhält. Die untere Reihe zeigt das Verhalten im Phasenraum, wo die exakte Lösung einem Kreis entspricht. Das Euler-Verfahren spiralt nach außen, was die künstliche Energiezufuhr widerspiegelt, während das Heun-Verfahren dem Kreis deutlich besser folgt.

i Vergleich mit dem Leapfrog-Verfahren

Das Heun-Verfahren und das in **?@sec-ch11-leapfrog** eingeführte Leapfrog-Verfahren sind beide Verfahren zweiter Ordnung mit Prädiktor-Korrektor-Struktur, unterscheiden sich aber grundlegend in ihrer Konzeption:

- **Runge-Kutta-Verfahren** (wie Heun) sind *universelle* Integratoren für beliebige Differentialgleichungen erster Ordnung $\mathbf{y}' = \mathbf{g}(\mathbf{y}, t)$. Sie machen keine Annahmen über die Struktur der Gleichung und sind daher breit einsetzbar.
- Das **Leapfrog-Verfahren** ist ein *spezialisierter* Integrator für Gleichungen der Form $\ddot{x} = f(x)$, bei denen die Beschleunigung nur von der Position abhängt. Diese Struktur ist charakteristisch für konservative mechanische Systeme (Hamilton-Systeme).

Der entscheidende Vorteil des Leapfrog-Verfahrens liegt in seiner Symplektizität: Es erhält das Phasenraumvolumen exakt und zeigt daher keine systematische Energiedrift bei Langzeitintegrationen. Das Heun-Verfahren ist *nicht* symplektisch – wie in Abbildung 1 sichtbar, spiralt auch die Heun-Trajektorie im Phasenraum, wenn auch langsamer als beim Euler-Verfahren.
Für Hamiltonsche Systeme ist daher das Leapfrog-Verfahren oft die bessere Wahl, während Runge-Kutta-Verfahren ihre Stärke bei allgemeinen, nicht-konservativen Problemen ausspielen.

Das klassische Runge-Kutta-Verfahren vierter Ordnung

Das am weitesten verbreitete Runge-Kutta-Verfahren ist das klassische Verfahren vierter Ordnung, das vier Funktionsauswertungen pro Zeitschritt verwendet. Es lässt sich als gewichteter Mittelwert von vier Steigungsschätzungen schreiben:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{\Delta t}{6}\left[\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right] \tag{7}$$

Die vier Steigungen werden dabei sequentiell berechnet, wobei jede nachfolgende Steigung auf den vorherigen aufbaut:

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{g}(\mathbf{y}_i, t_i) \\
\mathbf{k}_2 &= \mathbf{g}\left(\mathbf{y}_i + \frac{\Delta t}{2}\mathbf{k}_1, t_i + \frac{\Delta t}{2}\right) \\
\mathbf{k}_3 &= \mathbf{g}\left(\mathbf{y}_i + \frac{\Delta t}{2}\mathbf{k}_2, t_i + \frac{\Delta t}{2}\right) \\
\mathbf{k}_4 &= \mathbf{g}\left(\mathbf{y}_i + \Delta t\,\mathbf{k}_3, t_i + \Delta t\right)
\end{aligned} \tag{8}$$

Die erste Steigung $\mathbf{k}_1$ entspricht der Steigung am Anfang des Intervalls und ist identisch mit der beim Euler-Verfahren verwendeten Steigung. Die zweite Steigung $\mathbf{k}_2$ wird in der Mitte des Intervalls ausgewertet, wobei die Position durch einen halben Euler-Schritt mit $\mathbf{k}_1$ geschätzt wird. Die dritte Steigung $\mathbf{k}_3$ wird ebenfalls in der Mitte des Intervalls berechnet, verwendet aber die verbesserte Positionsschätzung basierend auf $\mathbf{k}_2$. Schließlich liefert $\mathbf{k}_4$ die Steigung am Ende des Intervalls unter Verwendung eines vollen Schritts mit $\mathbf{k}_3$.

Die Gewichte 1/6, 2/6, 2/6 und 1/6 in der Kombinationsformel sind nicht willkürlich gewählt, sondern so bestimmt, dass der Fehler optimal minimiert wird. Eine Taylorentwicklung zeigt, dass diese Koeffizienten zu einem lokalen Abbruchfehler von $\mathcal{O}(\Delta t^5)$ führen, was einem Verfahren vierter Ordnung entspricht.

Die Simpson-Regel

Die Gewichtung $(1+2+2+1)/6$ im klassischen RK4-Verfahren ist eng verwandt mit der Simpson-Regel für numerische Integration. Die Simpson-Regel approximiert ein Integral über das Intervall $[a, b]$ durch

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

Diese Formel gewichtet den Funktionswert in der Mitte des Intervalls vierfach so stark wie die Randwerte, was einer parabolischen Interpolation entspricht. Im RK4-Verfahren tragen die beiden Mittelwerte $\mathbf{k}_2$ und $\mathbf{k}_3$ zusammen das Gewicht $4/6$, während die Randwerte $\mathbf{k}_1$ und $\mathbf{k}_4$ je das Gewicht $1/6$ erhalten. Die Verwendung zweier Mittelpunktschätzungen statt einer einzigen ermöglicht es, den Fehler weiter zu reduzieren.

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

def rk4_step(g, y, t, dt):
    """Ein Schritt des klassischen RK4-Verfahrens"""
    k1 = g(y, t)
    k2 = g(y + dt/2 * k1, t + dt/2)
    k3 = g(y + dt/2 * k2, t + dt/2)
    k4 = g(y + dt * k3, t + dt)
    return y + dt/6 * (k1 + 2*k2 + 2*k3 + k4)

def integrate(step_func, g, y0, t_span, dt):
    """Integriere eine DGL"""
    t_start, t_end = t_span
    n_steps = int((t_end - t_start) / dt)
    t = np.linspace(t_start, t_end, n_steps + 1)
    y = np.zeros((len(t), len(y0)))
    y[0] = y0

    for i in range(len(t) - 1):
        y[i+1] = step_func(g, y[i], t[i], dt)

    return t, y

# Testproblem: exponentieller Zerfall
def exponential_decay(y, t):
    return -y
```

```
y0 = np.array([1.0])
t_end = 5.0

# Fehleranalyse
dt_values = np.logspace(-3, -0.5, 20)
errors_euler = []
errors_heun = []
errors_rk4 = []

for dt in dt_values:
    t_e, y_e = integrate(euler_step, exponential_decay, y0, (0, t_end), dt)
    t_h, y_h = integrate(heun_step, exponential_decay, y0, (0, t_end), dt)
    t_r, y_r = integrate(rk4_step, exponential_decay, y0, (0, t_end), dt)

    y_exact = np.exp(-t_end)

    errors_euler.append(np.abs(y_e[-1, 0] - y_exact))
    errors_heun.append(np.abs(y_h[-1, 0] - y_exact))
    errors_rk4.append(np.abs(y_r[-1, 0] - y_exact))

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

# Konvergenzplot
ax1.loglog(dt_values, errors_euler, 'ro-', markersize=4, label='Euler (Ordnung 1)')
ax1.loglog(dt_values, errors_heun, 'go-', markersize=4, label='Heun (Ordnung 2)')
ax1.loglog(dt_values, errors_rk4, 'bo-', markersize=4, label='RK4 (Ordnung 4)')

ax1.loglog(dt_values, 0.5*dt_values, 'r--', alpha=0.5, label=r'$\mathcal{O}(\Delta t)$')
ax1.loglog(dt_values, 0.1*dt_values**2, 'g--', alpha=0.5, label=r'$\mathcal{O}(\Delta t^2)$'
ax1.loglog(dt_values, 0.01*dt_values**4, 'b--', alpha=0.5, label=r'$\mathcal{O}(\Delta t^4)$

ax1.set_xlabel('Zeitschritt $\\Delta t$')
ax1.set_ylabel('Globaler Fehler')
ax1.set_title('Konvergenzordnung der Verfahren')
ax1.legend(fontsize=9)
ax1.grid(True, alpha=0.3)
ax1.set_xlim([1e-3, 0.5])
ax1.set_ylim([1e-14, 1])

# Effizienz
n_evals_euler = (t_end / dt_values).astype(int)
n_evals_heun = 2 * n_evals_euler
```

```
n_evals_rk4 = 4 * n_evals_euler

ax2.loglog(n_evals_euler, errors_euler, 'ro-', markersize=4, label='Euler')
ax2.loglog(n_evals_heun, errors_heun, 'go-', markersize=4, label='Heun')
ax2.loglog(n_evals_rk4, errors_rk4, 'bo-', markersize=4, label='RK4')

ax2.set_xlabel('Anzahl Funktionsauswertungen')
ax2.set_ylabel('Globaler Fehler')
ax2.set_title('Effizienz: Fehler vs. Rechenaufwand')
ax2.legend()
ax2.grid(True, alpha=0.3)

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

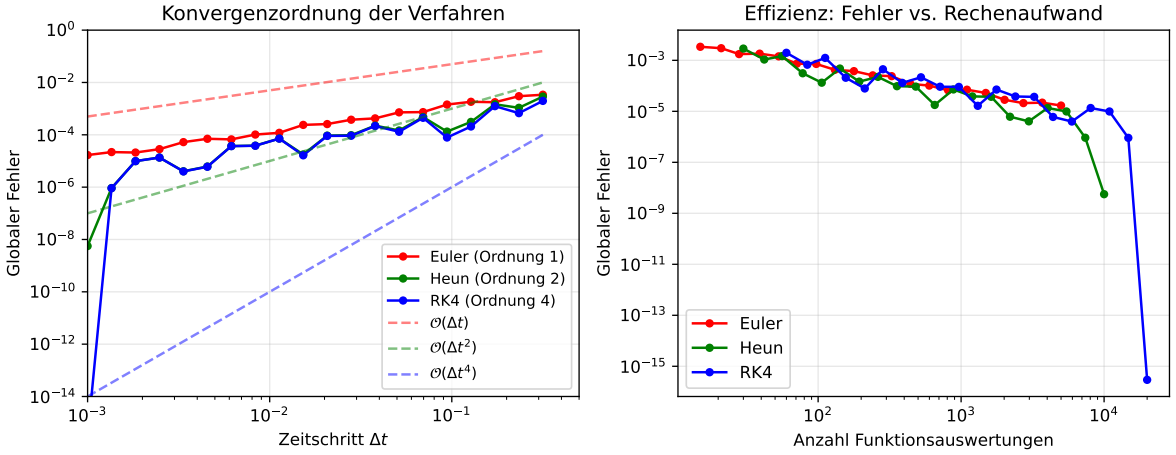

Abbildung 2: Konvergenzverhalten verschiedener Runge-Kutta-Verfahren. Das RK4-Verfahren erreicht bei gleichem Rechenaufwand eine deutlich höhere Genauigkeit als Verfahren niedrigerer Ordnung.

Abbildung 2 quantifiziert die Überlegenheit höherer Ordnungsverfahren. Der linke Teil zeigt die Konvergenzordnung: Der Fehler des Euler-Verfahrens skaliert linear mit Δt, der des Heun-Verfahrens quadratisch, und der des RK4-Verfahrens mit der vierten Potenz. Eine Halbierung des Zeitschritts reduziert beim RK4-Verfahren den Fehler um den Faktor 16.

Der rechte Teil betrachtet die Effizienz unter Berücksichtigung des Rechenaufwands. Obwohl das RK4-Verfahren viermal so viele Funktionsauswertungen pro Zeitschritt benötigt wie das Euler-Verfahren, erreicht es bei gleichem Gesamtaufwand eine um Größenordnungen höhere Genauigkeit. Dies macht höherwertige Verfahren in der Praxis nahezu immer vorzuziehen.

Die Funktion `solve_ivp`

Die Funktion `scipy.integrate.solve_ivp` stellt eine moderne, robuste Implementierung verschiedener Runge-Kutta-Verfahren mit automatischer Schrittweitenkontrolle bereit. Sie ist der empfohlene Standardweg zur numerischen Lösung gewöhnlicher Differentialgleichungen in Python.

Die verfügbaren Methoden umfassen `RK23`, ein eingebettetes Runge-Kutta-Verfahren der Ordnung 2 mit Fehlerschätzer der Ordnung 3, das für weniger anspruchsvolle Probleme geeignet ist. Die Standardmethode `RK45` ist ein Verfahren der Ordnung 4 mit Fehlerschätzer der Ordnung 5, bekannt als Dormand-Prince-Verfahren. Für Probleme, die höchste Genauigkeit erfordern, steht `DOP853` zur Verfügung, ein Verfahren der Ordnung 8 mit eingebetteten Fehlerschätzern der Ordnungen 5 und 3.

Die Notation "4(5)" bedeutet, dass das Verfahren selbst die Ordnung 4 hat, aber einen eingebetteten Fehlerschätzer der Ordnung 5 verwendet. Dieser Fehlerschätzer ermöglicht die automatische Anpassung des Zeitschritts, die im nächsten Kapitel behandelt wird.

Struktur eines `solve_ivp`-Aufrufs

Die Verwendung von `solve_ivp` folgt einem einheitlichen Muster. Die zu lösende Differentialgleichung wird als Funktion definiert, die den aktuellen Zeitpunkt und Zustand als Argumente erhält und die Zeitableitung zurückgibt:

```
from scipy.integrate import solve_ivp

def differentialgleichung(t, y, param1, param2):
    """
    Rechte Seite der DGL dy/dt = f(t, y)

    WICHTIG: Der Zeitpunkt t ist das erste Argument,
    der Zustandsvektor y das zweite!
    """
    dydt = ...  # Berechne Ableitungen
    return dydt

solution = solve_ivp(
    differentialgleichung,       # Die DGL-Funktion
    t_span=(t_start, t_end),     # Zeitintervall
    y0=[y1_0, y2_0, ...],        # Anfangsbedingungen
    args=(param1, param2),       # Zusätzliche Parameter
    method='RK45',               # Integrationsverfahren
```

```
    t_eval=t_array,              # Zeitpunkte für Ausgabe
    dense_output=True,           # Interpolation zwischen Punkten
    rtol=1e-3,                   # Relative Toleranz
    atol=1e-6                    # Absolute Toleranz
)

# Zugriff auf die Lösung
t = solution.t        # Zeitpunkte
y = solution.y        # Lösungswerte (y[0] ist erste Komponente)
```

> **!** Argumentreihenfolge
>
> Bei `solve_ivp` lautet die Reihenfolge der Argumente `dgl(t, y, ...)`, also Zeit vor Zustand. Dies unterscheidet sich von manchen älteren Konventionen und der Notation in diesem Skript, wo wir `g(y, t)` schreiben. Die Wahl von SciPy hat praktische Gründe, da sie das Hinzufügen optionaler Parameter erleichtert.

Beispiel: Der gedämpfte harmonische Oszillator

Als Anwendungsbeispiel betrachten wir den gedämpften harmonischen Oszillator, der durch die Bewegungsgleichung

$$m\ddot{y} + c\dot{y} + ky = 0 \tag{9}$$

beschrieben wird. Hierbei bezeichnet m die Masse, c die Dämpfungskonstante und k die Federkonstante.

Dieses System zweiter Ordnung lässt sich durch Einführung der Geschwindigkeit $v = \dot{y}$ in ein System erster Ordnung umschreiben:

$$\dot{y} = v, \quad \dot{v} = -\frac{c}{m}v - \frac{k}{m}y. \tag{10}$$

Das Verhalten des Systems hängt vom Verhältnis der Dämpfung zur kritischen Dämpfung $c_{\text{krit}} = 2\sqrt{km}$ ab. Für $c < c_{\text{krit}}$ schwingt das System gedämpft, für $c = c_{\text{krit}}$ liegt der aperiodische Grenzfall vor, und für $c > c_{\text{krit}}$ ist das System überdämpft und kriecht ohne Schwingung zum Gleichgewicht zurück.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def damped_oscillator(t, y, m, c, k):
```

```
    """
    Gedämpfter harmonischer Oszillator

    Der Zustandsvektor y = [Position, Geschwindigkeit] beschreibt
    den momentanen mechanischen Zustand des Systems.
    """
    return [y[1], -(c/m)*y[1] - (k/m)*y[0]]

# Parameter
m = 1.0
k = 4.0  # ergibt omega_0 = 2
c_values = [0.0, 0.5, 2.0, 4.0]
labels = ['Ungedämpft', 'Schwach gedämpft', 'Kritisch gedämpft', 'Stark gedämpft']

# Anfangsbedingungen und Zeitintervall
y0 = [1.0, 0.0]
t_span = (0, 10)
t_eval = np.linspace(0, 10, 500)

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

colors = ['blue', 'green', 'orange', 'red']

for c, label, color in zip(c_values, labels, colors):
    sol = solve_ivp(
        damped_oscillator,
        t_span,
        y0,
        args=(m, c, k),
        method='RK45',
        t_eval=t_eval,
        dense_output=True
    )

    ax1.plot(sol.t, sol.y[0], color=color, linewidth=2, label=label)
    ax2.plot(sol.y[0], sol.y[1], color=color, linewidth=2, label=label)

ax1.set_xlabel('Zeit $t$')
ax1.set_ylabel('Position $y(t)$')
ax1.set_title('Zeitverlauf')
ax1.legend()
ax1.grid(True, alpha=0.3)
```

```python
ax1.axhline(y=0, color='k', linewidth=0.5)

ax2.set_xlabel('Position $y$')
ax2.set_ylabel('Geschwindigkeit $\\dot{y}$')
ax2.set_title('Phasenraum')
ax2.legend()
ax2.grid(True, alpha=0.3)
ax2.axhline(y=0, color='k', linewidth=0.5)
ax2.axvline(x=0, color='k', linewidth=0.5)

plt.tight_layout()
plt.show()

# Ausgabe der Integrationsstatistik
sol = solve_ivp(damped_oscillator, t_span, y0, args=(m, 0.5, k), method='RK45')
print(f"Schwach gedämpfter Fall (c=0.5):")
print(f"  Anzahl der Zeitschritte: {len(sol.t)}")
print(f"  Anzahl der Funktionsauswertungen: {sol.nfev}")
print(f"  Integration erfolgreich: {sol.success}")
```

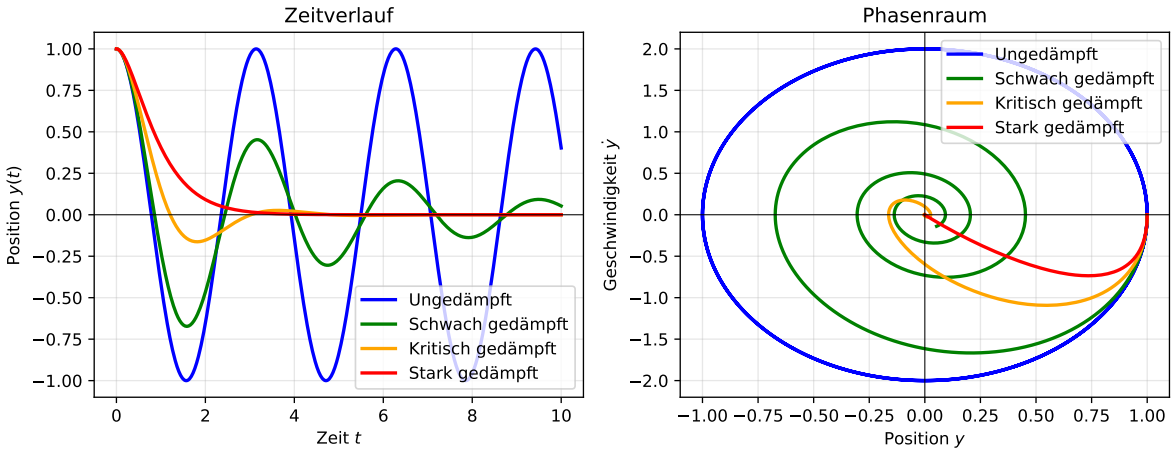

Abbildung 3: Lösung des gedämpften harmonischen Oszillators mit solve_ivp für verschiedene Dämpfungsstärken. Der Übergang vom schwingenden zum kriechenden Verhalten zeigt sich sowohl im Zeitverlauf als auch im Phasenraum.

```
Schwach gedämpfter Fall (c=0.5):
  Anzahl der Zeitschritte: 27
  Anzahl der Funktionsauswertungen: 170
  Integration erfolgreich: True
```

Abbildung 3 zeigt das charakteristische Verhalten für verschiedene Dämpfungsstärken. Im Zeitverlauf erkennt man, wie die Schwingungsamplitude bei schwacher Dämpfung langsam abnimmt, während bei kritischer und starker Dämpfung das System ohne Überschwingen zum Gleichgewicht zurückkehrt. Im Phasenraum manifestiert sich dies als einwärts spiralende Trajektorie bei schwacher Dämpfung und als direkter Weg zum Ursprung bei überkritischer Dämpfung.

Anwendung: Die Lotka-Volterra-Gleichungen

Die Lotka-Volterra-Gleichungen beschreiben die Dynamik eines idealisierten Räuber-Beute-Systems und sind ein klassisches Beispiel für ein nichtlineares Differentialgleichungssystem:

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= \alpha x - \beta xy \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= \delta xy - \gamma y \end{aligned} \tag{11}$$

In diesem System bezeichnet x die Beutepopulation und y die Räuberpopulation. Der Parameter α beschreibt die natürliche Wachstumsrate der Beute in Abwesenheit von Räubern, β die Rate, mit der Räuber Beute fressen, δ die Effizienz, mit der gefressene Beute in Räubernachwuchs umgewandelt wird, und γ die natürliche Sterberate der Räuber in Abwesenheit von Beute.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def lotka_volterra(t, z, alpha, beta, delta, gamma):
    """
    Lotka-Volterra-Gleichungen (Räuber-Beute-Modell)

    Der Zustandsvektor z = [x, y] enthält die Beute- und
    Räuberpopulation.
    """
    x, y = z
    dxdt = alpha * x - beta * x * y
    dydt = delta * x * y - gamma * y
    return [dxdt, dydt]

# Parameter
alpha = 1.0   # Wachstumsrate Beute
beta = 0.1    # Fressrate
delta = 0.075 # Wachstumseffizienz Räuber
```

```
gamma = 1.5   # Sterberate Räuber

# Anfangsbedingungen und Zeitintervall
z0 = [10.0, 5.0]
t_span = (0, 50)
t_eval = np.linspace(0, 50, 1000)

# Lösung
sol = solve_ivp(
    lotka_volterra,
    t_span,
    z0,
    args=(alpha, beta, delta, gamma),
    method='RK45',
    t_eval=t_eval
)

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

# Zeitverlauf
ax1.plot(sol.t, sol.y[0], 'b-', linewidth=2, label='Beute $x(t)$')
ax1.plot(sol.t, sol.y[1], 'r-', linewidth=2, label='Räuber $y(t)$')
ax1.set_xlabel('Zeit $t$')
ax1.set_ylabel('Population')
ax1.set_title('Populationsdynamik')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Phasenraum
ax2.plot(sol.y[0], sol.y[1], 'g-', linewidth=2)
ax2.plot(z0[0], z0[1], 'ko', markersize=10, label='Anfang')
ax2.set_xlabel('Beute $x$')
ax2.set_ylabel('Räuber $y$')
ax2.set_title('Phasenraum')
ax2.legend()
ax2.grid(True, alpha=0.3)

# Gleichgewichtspunkt markieren
x_eq = gamma / delta
y_eq = alpha / beta
ax2.plot(x_eq, y_eq, 'r*', markersize=15, label=f'Gleichgewicht ({x_eq:.1f}, {y_eq:.1f})')
ax2.legend()
```

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

print(f"Anzahl der adaptiven Zeitschritte: {len(sol.t)}")
print(f"Anzahl der Funktionsauswertungen: {sol.nfev}")
```

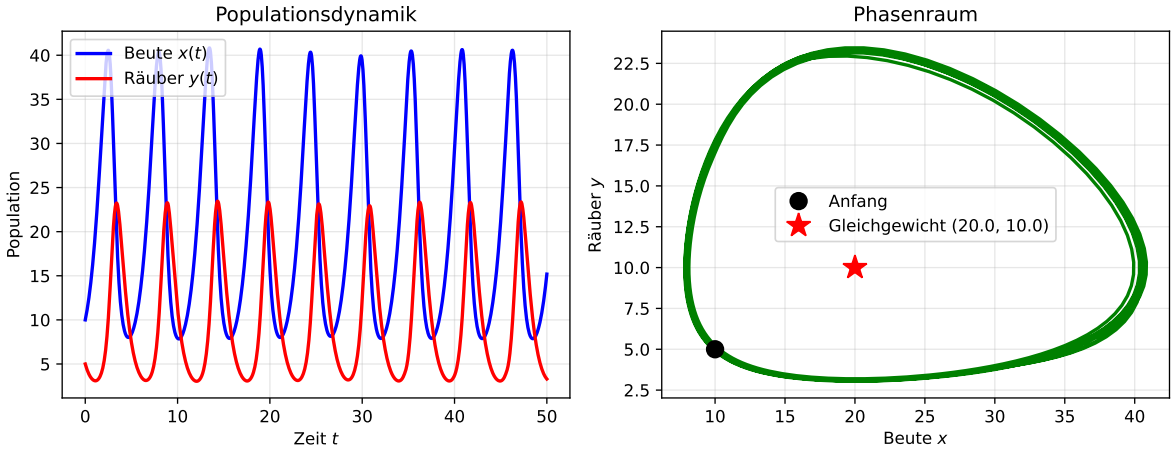

Abbildung 4: Numerische Lösung der Lotka-Volterra-Gleichungen. Die Populationen oszillieren periodisch, wobei die Räuberpopulation der Beutepopulation mit einer Phasenverschiebung folgt. Im Phasenraum zeigt sich die periodische Natur als geschlossene Kurve um den Gleichgewichtspunkt.

```
Anzahl der adaptiven Zeitschritte: 1000
Anzahl der Funktionsauswertungen: 554
```

Abbildung 4 zeigt die charakteristische Dynamik des Räuber-Beute-Systems. Im Zeitverlauf oszillieren beide Populationen periodisch, wobei die Maxima der Räuberpopulation den Maxima der Beutepopulation zeitversetzt folgen. Diese Phasenverschiebung hat eine intuitive Interpretation: Wenn viel Beute vorhanden ist, vermehren sich die Räuber, was zu einem Anstieg der Räuberpopulation führt. Die wachsende Räuberzahl dezimiert die Beute, woraufhin auch die Räuber aufgrund von Nahrungsmangel abnehmen. Mit weniger Räubern kann sich die Beute wieder erholen, und der Zyklus beginnt von neuem.

Im Phasenraum manifestiert sich diese periodische Dynamik als geschlossene Kurve um den Gleichgewichtspunkt. Die Tatsache, dass die Kurve geschlossen ist, zeigt die periodische Natur der Lösung. Das Gleichgewicht selbst ist instabil und wird nie erreicht; stattdessen führt jede Abweichung davon zu den beobachteten Oszillationen.

Zusammenfassung

Die Runge-Kutta-Verfahren bilden eine Familie von Integrationsverfahren, die durch Verwendung mehrerer Funktionsauswertungen pro Zeitschritt eine höhere Genauigkeit erreichen als das einfache Euler-Verfahren. Das Euler-Verfahren selbst kann als Runge-Kutta-Verfahren erster Ordnung betrachtet werden.

Das Heun-Verfahren oder Runge-Kutta-Verfahren zweiter Ordnung verwendet eine Prädiktor-Korrektor-Strategie mit zwei Funktionsauswertungen und erreicht quadratische Konvergenz. Das klassische Runge-Kutta-Verfahren vierter Ordnung kombiniert vier Steigungsschätzungen nach dem Prinzip der Simpson-Regel und erzielt bei moderatem Mehraufwand eine dramatisch bessere Genauigkeit.

Die Funktion `scipy.integrate.solve_ivp` bietet eine robuste Implementierung dieser Verfahren mit automatischer Schrittweitenkontrolle. Die Standardmethode `RK45` ist für die meisten Anwendungen geeignet und kombiniert gute Genauigkeit mit adaptiver Zeitschrittsteuerung.

> **Tip**
>
> Für die meisten praktischen Anwendungen ist `solve_ivp` mit der Standardmethode `RK45` eine ausgezeichnete Wahl. Die automatische Schrittweitenkontrolle sorgt für Effizienz und Zuverlässigkeit, und die Konvergenzordnung 4 ermöglicht hohe Genauigkeit bei moderatem Rechenaufwand.

Im nächsten Kapitel werden wir die automatische Zeitschrittkontrolle genauer untersuchen, die das zentrale Feature moderner Differentialgleichungslöser darstellt und `solve_ivp` so robust macht.