Zeitschrittkontrolle

Lernziele

Die Studierenden sollen…

- … das Konzept der automatischen Zeitschrittkontrolle beschreiben können.
- … das Prinzip eingebetteter Runge-Kutta-Verfahren zur Fehlerschätzung erklären können.
- … den Zusammenhang zwischen Toleranzparametern und Genauigkeit erklären können.
- … ein einfaches Runge-Kutta-Verfahren mit adaptiver Schrittweite implementieren können.
- … die Toleranzparameter von `solve_ivp` sinnvoll wählen können.

Einführung

In den vorherigen Kapiteln haben wir Runge-Kutta-Verfahren mit konstantem Zeitschritt Δt betrachtet. Für viele praktische Anwendungen ist ein fester Zeitschritt jedoch ineffizient oder sogar problematisch. Wenn die Lösung sich zeitweise schnell und zeitweise langsam ändert, müsste ein konstanter Zeitschritt so klein gewählt werden, dass er auch in den schnellen Phasen hinreichend genau ist. In den langsamen Phasen wäre dieser kleine Zeitschritt aber unnötig und verschwenderisch.

Die automatische Zeitschrittkontrolle löst dieses Problem, indem sie den Zeitschritt dynamisch an die lokale Dynamik der Lösung anpasst. In Phasen schneller Änderung werden kleine Zeitschritte verwendet, in Phasen langsamer Dynamik hingegen große. Dies führt zu einer optimalen Balance zwischen Genauigkeit und Rechenaufwand.

Das zentrale Element der Zeitschrittkontrolle ist die Fehlerschätzung. Um den Zeitschritt sinnvoll anpassen zu können, muss der Algorithmus abschätzen, wie groß der Fehler im aktuellen

Schritt ist. Eingebettete Runge-Kutta-Verfahren ermöglichen diese Schätzung auf elegante Weise, ohne zusätzliche Funktionsauswertungen zu erfordern.

Das Prinzip eingebetteter Verfahren

Die Grundidee eingebetteter Runge-Kutta-Verfahren besteht darin, aus denselben Funktionsauswertungen zwei Lösungsschätzungen unterschiedlicher Ordnung zu berechnen. Die Differenz dieser beiden Schätzungen liefert eine Approximation des lokalen Fehlers.

Betrachten wir ein Verfahren der Ordnung p mit eingebettetem Verfahren der Ordnung $p-1$. Beide Verfahren verwenden dieselben Steigungen $\mathbf{k}_1, \mathbf{k}_2, \dots, \mathbf{k}_s$, kombinieren sie aber mit unterschiedlichen Gewichten:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \sum_{i=1}^{s} b_i \mathbf{k}_i, \quad \hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \Delta t \sum_{i=1}^{s} \hat{b}_i \mathbf{k}_i. \tag{1}$$

Die höherwertige Lösung $\mathbf{y}_{n+1}$ wird für die Zeitpropagation verwendet, während die Differenz

$$\mathbf{e} = \mathbf{y}_{n+1} - \hat{\mathbf{y}}_{n+1} = \Delta t \sum_{i=1}^{s} (b_i - \hat{b}_i) \mathbf{k}_i \tag{2}$$

als Schätzung des lokalen Fehlers dient.

Diese Fehlerschätzung hat die Ordnung des niedrigeren Verfahrens, also $p-1$. Da der tatsächliche Fehler des höheren Verfahrens von Ordnung p ist, überschätzt die Fehlerschätzung den wahren Fehler leicht, was zu einer konservativen Schrittweitensteuerung führt.

Das RK23-Verfahren

Das RK23-Verfahren, auch bekannt als Bogacki-Shampine-Verfahren, ist ein eingebettetes Runge-Kutta-Verfahren der Ordnung 2 mit einem Fehlerschätzer der Ordnung 3. Es verwendet vier Stufen, wobei die vierte Stufe für den nächsten Zeitschritt wiederverwendet werden kann, sodass effektiv nur drei neue Funktionsauswertungen pro Schritt benötigt werden.

Die Stufen des Verfahrens lauten:

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{g}(\mathbf{y}_n, t_n) \\
\mathbf{k}_2 &= \mathbf{g}\left(\mathbf{y}_n + \frac{\Delta t}{2}\mathbf{k}_1, t_n + \frac{\Delta t}{2}\right) \\
\mathbf{k}_3 &= \mathbf{g}\left(\mathbf{y}_n + \frac{3\Delta t}{4}\mathbf{k}_2, t_n + \frac{3\Delta t}{4}\right) \\
\mathbf{k}_4 &= \mathbf{g}(\mathbf{y}_{n+1}, t_{n+1})
\end{aligned} \tag{3}$$

Die Lösung dritter Ordnung (lokaler Fehler $\mathcal{O}(\Delta t^4)$) wird für die Zeitpropagation verwendet:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{9}(2\mathbf{k}_1 + 3\mathbf{k}_2 + 4\mathbf{k}_3) + \mathcal{O}(\Delta t^4) \tag{4}$$

Die eingebettete Lösung zweiter Ordnung (lokaler Fehler $\mathcal{O}(\Delta t^3)$) dient der Fehlerschätzung:

$$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{24}(7\mathbf{k}_1 + 6\mathbf{k}_2 + 8\mathbf{k}_3 + 3\mathbf{k}_4) + \mathcal{O}(\Delta t^3) \tag{5}$$

Der Fehlerschätzer ergibt sich aus der Differenz und ist von Ordnung $\mathcal{O}(\Delta t^3)$:

$$\mathbf{e} = \mathbf{y}_{n+1} - \hat{\mathbf{y}}_{n+1} = \frac{\Delta t}{72}(-5\mathbf{k}_1 + 6\mathbf{k}_2 + 8\mathbf{k}_3 - 9\mathbf{k}_4) + \mathcal{O}(\Delta t^4) \tag{6}$$

Da der Fehlerschätzer den Fehler der niedrigeren Ordnung ($\mathcal{O}(\Delta t^3)$) approximiert, während die verwendete Lösung einen kleineren Fehler ($\mathcal{O}(\Delta t^4)$) hat, ist die Schätzung konservativ.

Schrittweitensteuerung

Der geschätzte Fehler $\mathbf{e}$ wird gegen eine vom Benutzer spezifizierte Toleranz verglichen. Übliche Implementierungen verwenden eine Kombination aus relativer und absoluter Toleranz:

$$\epsilon_{\text{tol}} = \text{atol} + \text{rtol} \cdot |\mathbf{y}_{n+1}| \tag{7}$$

Der skalierte Fehler wird dann als

$$\epsilon = \left\| \frac{\mathbf{e}}{\epsilon_{\text{tol}}} \right\| \tag{8}$$

berechnet, wobei typischerweise die euklidische Norm oder die Maximumsnorm verwendet wird.

Wenn der skalierte Fehler ϵ kleiner als 1 ist, wird der Schritt akzeptiert. Andernfalls wird der Schritt verworfen und mit einer kleineren Schrittweite wiederholt. In beiden Fällen wird die Schrittweite für den nächsten Schritt angepasst.

Die optimale neue Schrittweite ergibt sich aus der Überlegung, dass der Fehler eines Verfahrens der Ordnung p wie Δt^{p+1} skaliert. Wenn der aktuelle Fehler ϵ ist und wir einen Fehler von etwa 1 erreichen wollen, sollte die neue Schrittweite sein:

$$\Delta t_{\text{neu}} = \Delta t_{\text{alt}} \cdot \left(\frac{1}{\epsilon}\right)^{1/(p+1)} = \Delta t_{\text{alt}} \cdot \epsilon^{-1/(p+1)} \tag{9}$$

In der Praxis wird diese Formel oft mit einem Sicherheitsfaktor $S < 1$ (typisch $S = 0.9$) multipliziert, um zu vermeiden, dass der Schritt häufig abgelehnt wird:

$$\Delta t_{\text{neu}} = S \cdot \Delta t_{\text{alt}} \cdot \epsilon^{-1/(p+1)} \tag{10}$$

Implementierung eines RK23-Verfahrens mit Schrittweitensteuerung

Die folgende Implementierung zeigt ein vollständiges RK23-Verfahren mit adaptiver Schrittweite. Das Verfahren implementiert die oben beschriebenen Formeln und demonstriert die wesentlichen Komponenten einer adaptiven Zeitintegration.

Als Testproblem verwenden wir den **Van-der-Pol-Oszillator**, einen klassischen nichtlinearen Oszillator mit der Bewegungsgleichung

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0. \tag{11}$$

Der Parameter μ steuert die Nichtlinearität. Für $\mu > 0$ zeigt der Oszillator Relaxationsoszillationen: Bei kleinen Amplituden ($|x| < 1$) wirkt eine negative Dämpfung, die das System antreibt, während bei großen Amplituden ($|x| > 1$) eine positive Dämpfung bremst. Dies führt zu einem stabilen Grenzzyklus.

Als System erster Ordnung mit $v = \dot{x}$ geschrieben:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= \mu(1 - x^2)v - x \end{aligned} \tag{12}$$

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

def rk23_adaptive(g, y0, t_span, rtol=1e-3, atol=1e-6, dt_init=None):
    """
    Runge-Kutta 2(3) Verfahren mit adaptiver Schrittweite

    Das Verfahren verwendet ein eingebettetes Paar der Ordnungen 2 und 3
    zur Fehlerschätzung und passt die Schrittweite automatisch an.

    Parameter:
    ----------
    g : callable
        Rechte Seite der DGL, g(y, t)
    y0 : array
        Anfangsbedingung
    t_span : tuple
        Zeitintervall (t_start, t_end)
    rtol : float
        Relative Toleranz
    atol : float
        Absolute Toleranz
    dt_init : float
```

```
        Anfangsschrittweite (optional)

    Rückgabe:
    ---------
    t_list : array
        Zeitpunkte
    y_list : array
        Lösungswerte
    dt_list : array
        Verwendete Schrittweiten
    err_list : array
        Geschätzte lokale Fehler
    """
    t_start, t_end = t_span
    y = np.array(y0, dtype=float)
    t = t_start

    # Anfangsschrittweite
    if dt_init is None:
        dt = (t_end - t_start) / 100
    else:
        dt = dt_init

    # Sicherheitsfaktor für Schrittweitenänderung
    safety = 0.9
    # Maximale Schrittweitenänderung pro Schritt
    max_factor = 5.0
    min_factor = 0.2

    # Speicher für Ergebnisse
    t_list = [t]
    y_list = [y.copy()]
    dt_list = []
    err_list = []

    while t < t_end:
        # Begrenze Schrittweite auf verbleibendes Intervall
        dt = min(dt, t_end - t)

        # RK23 Stufen (Bogacki-Shampine)
        k1 = g(y, t)
        k2 = g(y + 0.5 * dt * k1, t + 0.5 * dt)
```

```
        k3 = g(y + 0.75 * dt * k2, t + 0.75 * dt)

        # Lösung dritter Ordnung
        y_new = y + dt / 9 * (2 * k1 + 3 * k2 + 4 * k3)

        # Vierte Stufe für Fehlerschätzung
        k4 = g(y_new, t + dt)

        # Fehlerschätzung
        error = dt / 72 * (-5 * k1 + 6 * k2 + 8 * k3 - 9 * k4)

        # Skalierter Fehler
        scale = atol + rtol * np.maximum(np.abs(y), np.abs(y_new))
        err_norm = np.sqrt(np.mean((error / scale)**2))

        if err_norm <= 1.0:
            # Schritt akzeptiert
            t = t + dt
            y = y_new
            t_list.append(t)
            y_list.append(y.copy())
            dt_list.append(dt)
            err_list.append(err_norm)

        # Neue Schrittweite berechnen (Ordnung p=2 des Hauptverfahrens)
        if err_norm == 0:
            factor = max_factor
        else:
            factor = safety * err_norm**(-1/3)

        factor = max(min_factor, min(max_factor, factor))
        dt = dt * factor

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

# Testproblem: Van-der-Pol-Oszillator
# Ein nichtlinearer Oszillator mit phasenabhängiger Dynamik
def van_der_pol(y, t, mu=1.0):
    """
    Van-der-Pol-Oszillator
```

```
    Der Van-der-Pol-Oszillator zeigt nichtlineare Relaxationsoszillationen.
    Bei kleinen Amplituden verhält er sich wie ein gedämpfter Oszillator,
    bei großen Amplituden wie ein getriebenes System.
    """
    x, v = y
    return np.array([v, mu * (1 - x**2) * v - x])

# Integration
t_span = (0, 30)
y0 = [2.0, 0.0]

t, y, dt_history, err_history = rk23_adaptive(
    lambda y, t: van_der_pol(y, t, mu=2.0),
    y0,
    t_span,
    rtol=1e-4,
    atol=1e-7
)

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

# Lösung im Zeitbereich
axes[0, 0].plot(t, y[:, 0], 'b-', linewidth=1.5, label='$x(t)$')
axes[0, 0].plot(t, y[:, 1], 'r--', linewidth=1.5, alpha=0.7, label='$v(t)$')
axes[0, 0].set_xlabel('Zeit $t$')
axes[0, 0].set_ylabel('Zustand')
axes[0, 0].set_title('Van-der-Pol-Oszillator')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# Phasenraum
axes[0, 1].plot(y[:, 0], y[:, 1], 'g-', linewidth=1)
axes[0, 1].plot(y0[0], y0[1], 'ko', markersize=8, label='Start')
axes[0, 1].set_xlabel('Position $x$')
axes[0, 1].set_ylabel('Geschwindigkeit $v$')
axes[0, 1].set_title('Phasenraum')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# Adaptive Schrittweite
t_mid = 0.5 * (t[:-1] + t[1:])
axes[1, 0].semilogy(t_mid, dt_history, 'b-', linewidth=1)
```

```python
axes[1, 0].set_xlabel('Zeit $t$')
axes[1, 0].set_ylabel('Schrittweite $\\Delta t$')
axes[1, 0].set_title('Adaptive Schrittweite')
axes[1, 0].grid(True, alpha=0.3)

# Geschätzter Fehler
axes[1, 1].semilogy(t_mid, err_history, 'r-', linewidth=1)
axes[1, 1].axhline(y=1, color='k', linestyle='--', alpha=0.5,
                    label='Toleranzgrenze')
axes[1, 1].set_xlabel('Zeit $t$')
axes[1, 1].set_ylabel('Skalierter Fehler')
axes[1, 1].set_title('Lokaler Fehlerschätzer')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Anzahl der Zeitschritte: {len(t) - 1}")
print(f"Minimale Schrittweite: {np.min(dt_history):.6f}")
print(f"Maximale Schrittweite: {np.max(dt_history):.6f}")
print(f"Verhältnis max/min: {np.max(dt_history) / np.min(dt_history):.1f}")
```

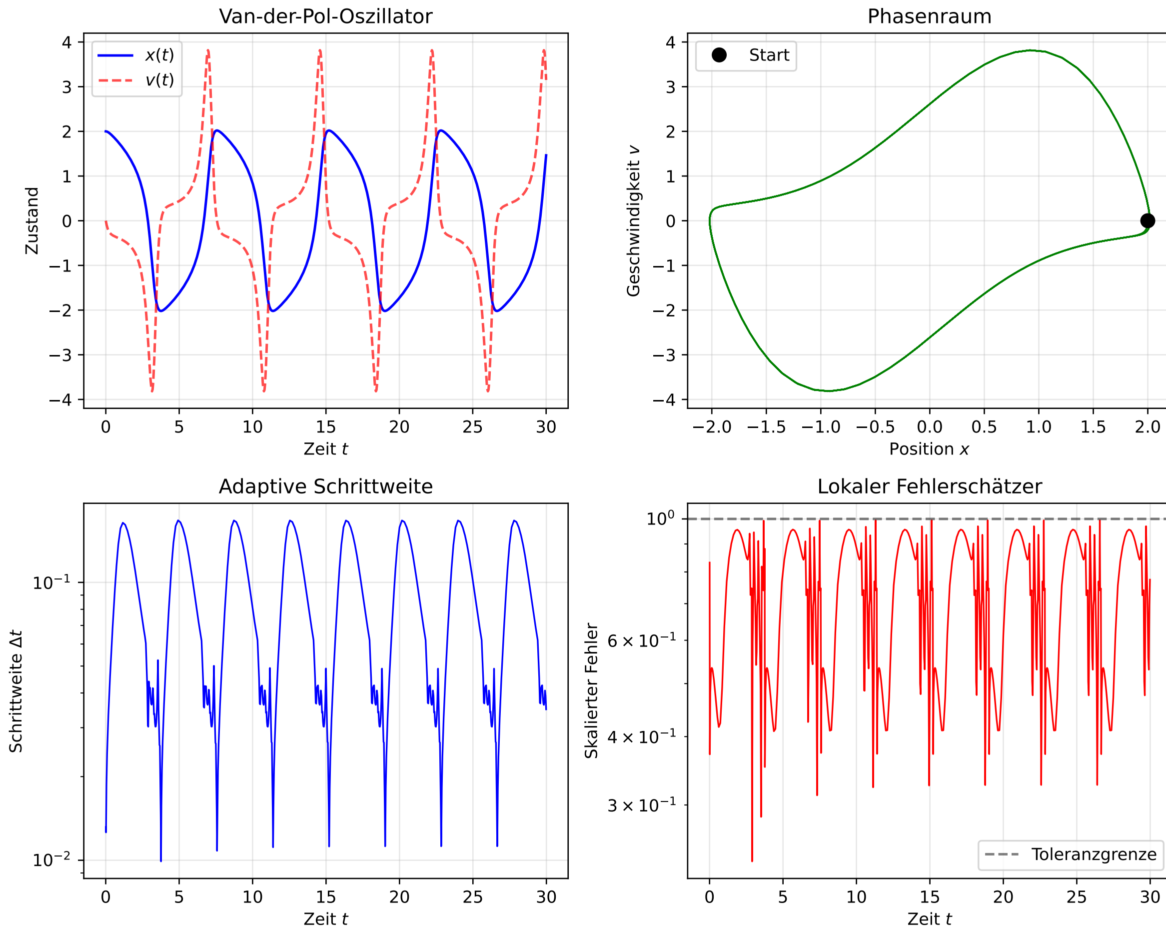

Abbildung 1: Demonstration des selbstimplementierten RK23-Verfahrens mit adaptiver Schrittweite für einen nichtlinearen Oszillator. Die obere Reihe zeigt die Lösung und den Phasenraum, die untere Reihe die adaptiv gewählte Schrittweite und den geschätzten lokalen Fehler.

```
Anzahl der Zeitschritte: 512
Minimale Schrittweite: 0.009904
Maximale Schrittweite: 0.167250
Verhältnis max/min: 16.9
```

Abbildung 1 zeigt das Verhalten des adaptiven RK23-Verfahrens am Beispiel des Van-der-Pol-Oszillators. Dieser nichtlineare Oszillator zeigt ausgeprägte Relaxationsoszillationen, bei denen auf langsame Phasen nahezu konstanter Position schnelle Übergänge folgen.

Die untere linke Grafik zeigt die adaptive Schrittweite. Man erkennt deutlich, wie das Verfahren in den schnellen Phasen die Schrittweite drastisch reduziert, während es in den langsamen

Phasen große Schritte verwendet. Diese Anpassung erfolgt vollautomatisch basierend auf dem geschätzten lokalen Fehler.

Die untere rechte Grafik zeigt den skalierten lokalen Fehler. Der Fehler bleibt durch die Schrittweitensteuerung stets unter der Toleranzgrenze von 1. In den schnellen Phasen liegt der Fehler nahe an dieser Grenze, was zeigt, dass das Verfahren die minimal nötige Schrittweite verwendet.

Vergleich mit `solve_ivp`

Die Funktion `scipy.integrate.solve_ivp` verwendet im Standardfall das RK45-Verfahren, ein eingebettetes Paar der Ordnungen 4 und 5. Das Prinzip der Schrittweitensteuerung ist jedoch dasselbe wie beim hier implementierten RK23. Der folgende Code vergleicht unsere Implementierung mit der professionellen Implementierung in SciPy.

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

# Gleiches Testproblem
def van_der_pol_scipy(t, y, mu=2.0):
    """Van-der-Pol für solve_ivp (t, y Reihenfolge)"""
    x, v = y
    return [v, mu * (1 - x**2) * v - x]

t_span = (0, 30)
y0 = [2.0, 0.0]
rtol, atol = 1e-4, 1e-7

# Eigene Implementierung
t_own, y_own, dt_own, _ = rk23_adaptive(
    lambda y, t: van_der_pol(y, t, mu=2.0),
    y0, t_span, rtol=rtol, atol=atol
)

# SciPy RK23
sol_rk23 = solve_ivp(
    van_der_pol_scipy, t_span, y0,
    method='RK23', rtol=rtol, atol=atol
)

# SciPy RK45 zum Vergleich
```

```python
sol_rk45 = solve_ivp(
    van_der_pol_scipy, t_span, y0,
    method='RK45', rtol=rtol, atol=atol
)

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

# Lösungsvergleich
axes[0].plot(t_own, y_own[:, 0], 'b-', linewidth=2,
             label='Eigene RK23', alpha=0.7)
axes[0].plot(sol_rk23.t, sol_rk23.y[0], 'r--', linewidth=2,
             label='SciPy RK23')
axes[0].plot(sol_rk45.t, sol_rk45.y[0], 'g:', linewidth=2,
             label='SciPy RK45')
axes[0].set_xlabel('Zeit $t$')
axes[0].set_ylabel('Position $x(t)$')
axes[0].set_title('Lösungsvergleich')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Schrittanzahl
methods = ['Eigene RK23', 'SciPy RK23', 'SciPy RK45']
steps = [len(t_own)-1, len(sol_rk23.t)-1, len(sol_rk45.t)-1]
evals = [4*(len(t_own)-1), sol_rk23.nfev, sol_rk45.nfev]

x = np.arange(len(methods))
width = 0.35

axes[1].bar(x - width/2, steps, width, label='Zeitschritte', color='blue', alpha=0.7)
axes[1].bar(x + width/2, evals, width, label='Fkt.-Auswertungen', color='orange', alpha=0.7)
axes[1].set_ylabel('Anzahl')
axes[1].set_title('Effizienzvergleich')
axes[1].set_xticks(x)
axes[1].set_xticklabels(methods, rotation=15)
axes[1].legend()
axes[1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()

print("Vergleich der Verfahren:")
print(f"  Eigene RK23: {len(t_own)-1} Schritte")
```

```
print(f"  SciPy RK23:  {len(sol_rk23.t)-1} Schritte, {sol_rk23.nfev} Auswertungen")
print(f"  SciPy RK45:  {len(sol_rk45.t)-1} Schritte, {sol_rk45.nfev} Auswertungen")
```

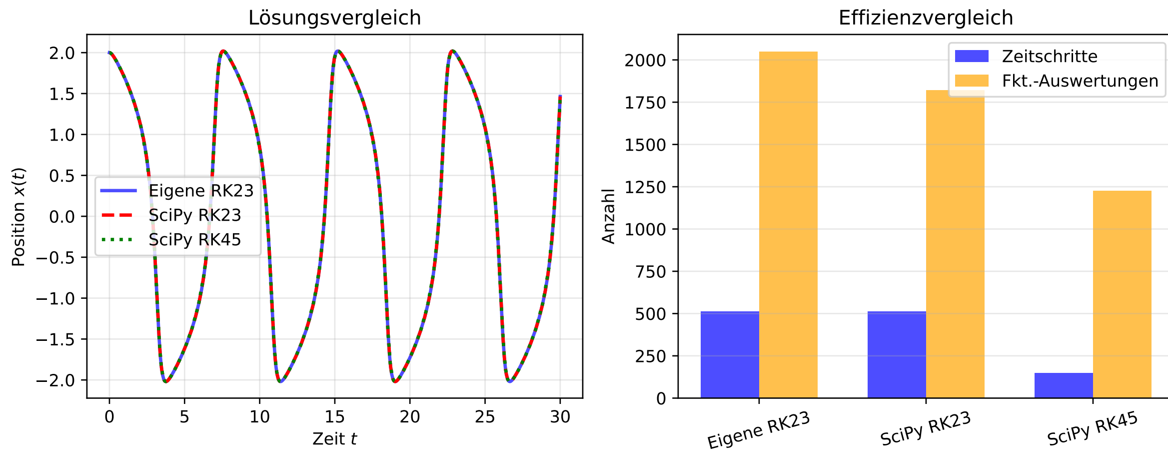

Abbildung 2: Vergleich der selbstimplementierten RK23-Methode mit solve_ivp. Beide Verfahren passen die Schrittweite in ähnlicher Weise an die lokale Dynamik an.

```
Vergleich der Verfahren:
  Eigene RK23: 512 Schritte
  SciPy RK23:  513 Schritte, 1820 Auswertungen
  SciPy RK45:  149 Schritte, 1226 Auswertungen
```

Abbildung 2 zeigt, dass unsere einfache Implementierung vergleichbare Ergebnisse wie die professionelle SciPy-Implementierung liefert. Die Lösungen stimmen gut überein, und die Anzahl der benötigten Schritte ist ähnlich. Das RK45-Verfahren benötigt weniger Schritte als RK23, da es durch seine höhere Ordnung größere Zeitschritte erlaubt.

Wahl der Toleranzparameter

Die Toleranzparameter `rtol` und `atol` kontrollieren die Genauigkeit der numerischen Lösung. Die relative Toleranz `rtol` bestimmt die relative Genauigkeit, während die absolute Toleranz `atol` die Genauigkeit für kleine Werte nahe null festlegt.

Die Wahl dieser Parameter hängt von der Anwendung ab. Für grobe Übersichtsrechnungen genügen oft Toleranzen von 10^{-3}, während präzise wissenschaftliche Berechnungen Toleranzen von 10^{-9} oder kleiner erfordern können. Unterhalb von etwa 10^{-14} dominieren Rundungsfehler der Gleitkommaarithmetik, sodass noch kleinere Toleranzen keine weitere Genauigkeitsverbesserung bringen.

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

# Einfaches Testproblem mit bekannter Lösung
def decay(t, y):
    return [-y[0]]

t_span = (0, 10)
y0 = [1.0]
y_exact_final = np.exp(-10)

# Verschiedene Toleranzen
tol_values = np.logspace(-2, -12, 20)
steps_list = []
errors_list = []
nfev_list = []

for tol in tol_values:
    sol = solve_ivp(decay, t_span, y0, method='RK45', rtol=tol, atol=tol/100)
    steps_list.append(len(sol.t) - 1)
    nfev_list.append(sol.nfev)
    error = np.abs(sol.y[0, -1] - y_exact_final)
    errors_list.append(error if error > 1e-16 else 1e-16)

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

# Aufwand vs. Toleranz
ax1.loglog(tol_values, steps_list, 'bo-', markersize=4, label='Zeitschritte')
ax1.loglog(tol_values, nfev_list, 'rs-', markersize=4, label='Fkt.-Auswertungen')
ax1.set_xlabel('Toleranz (rtol)')
ax1.set_ylabel('Anzahl')
ax1.set_title('Rechenaufwand')
ax1.legend()
ax1.grid(True, alpha=0.3)
ax1.invert_xaxis()

# Fehler vs. Toleranz
ax2.loglog(tol_values, errors_list, 'go-', markersize=4)
ax2.loglog(tol_values, tol_values, 'k--', alpha=0.5, label='Toleranz')
ax2.loglog(tol_values, np.ones_like(tol_values)*1e-15, 'r--', alpha=0.5,
           label='Maschinengenauigkeit')
```

```
ax2.set_xlabel('Toleranz (rtol)')
ax2.set_ylabel('Tatsächlicher Fehler')
ax2.set_title('Genauigkeit')
ax2.legend()
ax2.grid(True, alpha=0.3)
ax2.invert_xaxis()

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

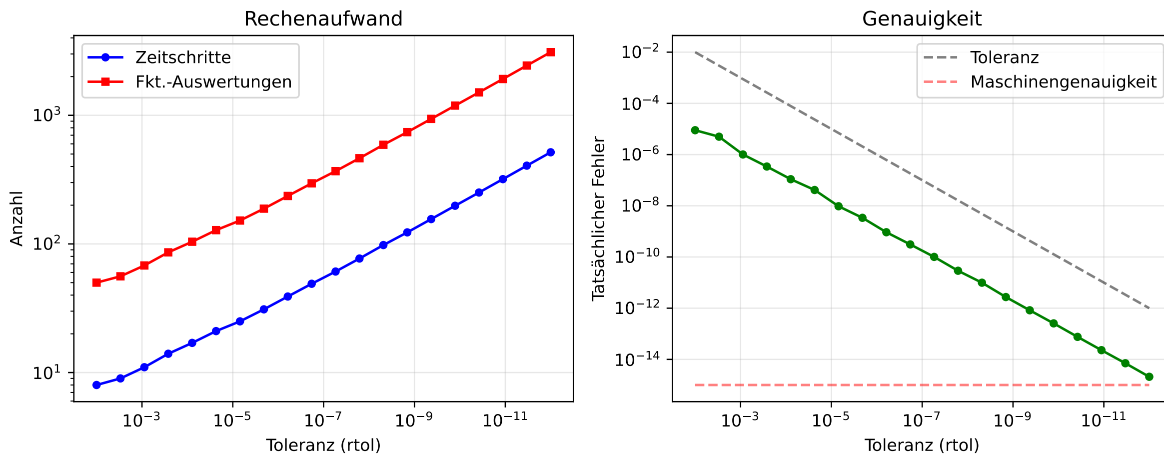

Abbildung 3: Einfluss der Toleranzwahl auf Genauigkeit und Rechenaufwand. Kleinere Toleranzen führen zu mehr Zeitschritten, aber auch zu genaueren Ergebnissen.

Abbildung 3 zeigt den Zusammenhang zwischen Toleranz und Aufwand beziehungsweise Genauigkeit. Der Rechenaufwand steigt mit sinkender Toleranz, da mehr Zeitschritte benötigt werden. Der tatsächliche Fehler folgt in etwa der angeforderten Toleranz, bis bei sehr kleinen Toleranzen die Maschinengenauigkeit erreicht wird.

Empfehlungen für die Toleranzwahl

Für schnelle Visualisierungen und explorative Rechnungen sind Toleranzen von `rtol=1e-3, atol=1e-6` meist ausreichend. Für Produktionsrechnungen empfehlen sich `rtol=1e-6, atol=1e-9`. Höchste Genauigkeit erreicht man mit `rtol=1e-10, atol=1e-13`, wobei man bedenken sollte, dass der Rechenaufwand dabei erheblich steigt.

Steife Differentialgleichungen

Manche Differentialgleichungen enthalten sehr unterschiedliche Zeitskalen. Solche Systeme werden als steif bezeichnet. Ein steifes System liegt vor, wenn die Eigenwerte der Jacobi-Matrix der rechten Seite betragsmäßig sehr unterschiedliche Werte haben.

Ein einfaches Beispiel ist das lineare System

$$\begin{aligned} \dot{y}_1 &= -1000\, y_1 + y_2 \\ \dot{y}_2 &= y_1 - y_2 \end{aligned} \tag{13}$$

Die Eigenwerte dieses Systems sind ungefähr $\lambda_1 \approx -1000$ und $\lambda_2 \approx -1$. Die schnelle Komponente klingt mit einer Zeitkonstante von etwa $\tau_1 = 0{,}001$ ab, während die langsame Komponente eine Zeitkonstante von $\tau_2 = 1$ hat. Dieses Verhältnis von $1000 : 1$ macht das System steif.

Für steife Probleme sind explizite Runge-Kutta-Verfahren wie RK23 oder RK45 oft ineffizient, da sie sehr kleine Zeitschritte benötigen, um stabil zu bleiben. In solchen Fällen sind implizite Verfahren wie `Radau` oder `BDF` vorzuziehen, die auch bei großen Zeitschritten stabil bleiben.

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

def stiff_system(t, y):
    """
    Ein steifes System mit zwei sehr unterschiedlichen Zeitskalen.
    Die schnelle Komponente klingt mit Zeitkonstante 0.001 ab,
    die langsame mit Zeitkonstante 1.
    """
    return [-1000*y[0] + y[1], y[0] - y[1]]

y0 = [1.0, 0.0]
t_span = (0, 1)
t_eval = np.linspace(0, 1, 200)

# Vergleich verschiedener Verfahren
methods = ['RK45', 'RK23', 'Radau', 'BDF']
colors = ['blue', 'green', 'red', 'purple']
results = {}

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

for method, color in zip(methods, colors):
    try:
```

```
        sol = solve_ivp(stiff_system, t_span, y0, method=method,
                       t_eval=t_eval, rtol=1e-6, atol=1e-9)
        results[method] = sol
        axes[0].plot(sol.t, sol.y[0], color=color, linewidth=1.5,
                    label=f'{method}')
    except Exception as e:
        print(f"{method} fehlgeschlagen: {e}")

axes[0].set_xlabel('Zeit $t$')
axes[0].set_ylabel('$y_1(t)$')
axes[0].set_title('Lösung (schnelle Komponente)')
axes[0].legend()
axes[0].grid(True, alpha=0.3)
axes[0].set_xlim([0, 0.05])

# Effizienzvergleich
method_names = []
nfev_values = []
for method in methods:
    if method in results:
        method_names.append(method)
        nfev_values.append(results[method].nfev)

axes[1].bar(method_names, nfev_values, color=['blue', 'green', 'red', 'purple'],
           alpha=0.7)
axes[1].set_ylabel('Funktionsauswertungen')
axes[1].set_title('Effizienzvergleich')
axes[1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()

print("Funktionsauswertungen:")
for method in methods:
    if method in results:
        print(f"  {method}: {results[method].nfev}")
```

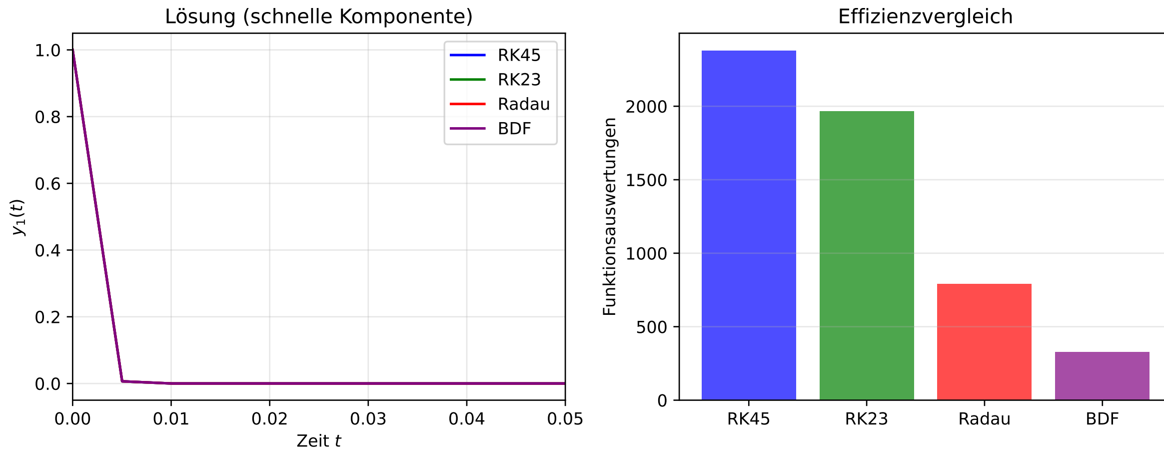

Abbildung 4: Vergleich von expliziten und impliziten Verfahren für ein steifes Problem. Das implizite Radau-Verfahren benötigt wesentlich weniger Funktionsauswertungen als das explizite RK45-Verfahren.

```
Funktionsauswertungen:
  RK45: 2378
  RK23: 1967
  Radau: 791
  BDF: 328
```

Abbildung 4 demonstriert den Unterschied zwischen expliziten und impliziten Verfahren für ein steifes Problem. Die impliziten Verfahren `Radau` und `BDF` benötigen deutlich weniger Funktionsauswertungen als die expliziten Verfahren `RK45` und `RK23`, obwohl alle dieselbe Genauigkeit erreichen.

> **Hinweis**
>
> Wenn ein Problem mit `RK45` ungewöhnlich viele Zeitschritte benötigt oder die Integration sehr langsam ist, könnte das Problem steif sein. In diesem Fall sollte man implizite Verfahren wie `method='Radau'` oder `method='BDF'` ausprobieren.

Zusammenfassung

Die automatische Zeitschrittkontrolle ist ein wesentliches Feature moderner Differentialgleichungslöser. Sie ermöglicht eine effiziente Integration, indem sie die Schrittweite automatisch an die lokale Dynamik anpasst.

Eingebettete Runge-Kutta-Verfahren wie RK23 oder RK45 berechnen aus denselben Funktionsauswertungen zwei Lösungsschätzungen unterschiedlicher Ordnung. Die Differenz dieser Schätzungen dient als lokaler Fehlerschätzer, der die Grundlage für die Schrittweitensteuerung bildet.

Die Toleranzparameter `rtol` und `atol` erlauben die Kontrolle über die Genauigkeit der Integration. Kleinere Toleranzen führen zu genaueren, aber rechenaufwändigeren Lösungen. Die Wahl der Toleranz sollte an die Anforderungen der jeweiligen Anwendung angepasst werden.

Für steife Probleme, die sehr unterschiedliche Zeitskalen enthalten, sind implizite Verfahren wie `Radau` oder `BDF` effizienter als explizite Runge-Kutta-Verfahren. Die Funktion `solve_ivp` bietet eine einheitliche Schnittstelle zu all diesen Verfahren.

> **!** Wichtig
>
> Die in diesem Kapitel entwickelte Implementierung eines RK23-Verfahrens mit Schrittweitensteuerung verdeutlicht die grundlegenden Prinzipien adaptiver Integratoren. Für praktische Anwendungen empfiehlt sich jedoch die Verwendung der professionell implementierten und ausgiebig getesteten Funktion `solve_ivp` aus SciPy.

Mit diesem Kapitel haben wir die wichtigsten Konzepte der numerischen Lösung gewöhnlicher Differentialgleichungen abgeschlossen. Die behandelten Verfahren bilden das Fundament für das Verständnis moderner Softwarebibliotheken und ermöglichen es, Differentialgleichungen effizient und zuverlässig zu lösen.