notebook_2_f

May 27, 2025

1 Notebook 2 - Symplektische Integratoren

Namen: Friedward Wenzler, Yueheng Li —> Erreichte Punktzahl: 10/10 +0,5 Bonus

Erweitern Sie Ihre conda Umgebung `python-science` um das Paket `sympy`. Aktivieren Sie dafür die virtuelle Umgebung `python-science` und laden Sie `sympy` mit:

```
conda install -c conda-forge sympy
```

Starten Sie anschließend Ihre jupyter-Instanz neu. Schließen Sie dazu alle offenen Fenster, beenden Sie die `jupyter-ServerApp` (z.B. mit `Strg+C`) und starten Sie anschließend Jupyter-Lab erneut.

Nun sollten die folgenden imports ohne Fehlermeldung ausgeführt werden.

[36]:
```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
import scipy.integrate
from scipy.optimize import fsolve
from ipywidgets import interact
import ipywidgets as widgets
```

2 Aufgabe 1 - Mathematisches Pendel

4/4 +0,5

Eine Masse m wird an einem masselosen Stab der Länge ℓ befestigt, welcher sich am Aufhängungspunkt frei bewegen kann. Die Beschleunigung durch Gravitation beträgt $g = 9,81 \frac{m}{s^2}$. Dieses mathematische Pendel (vgl. Beispiel 13.19) kann durch die Hamilton-Funktion

$$H(p,q) = \frac{1}{2m\ell^2}p^2 + m\ell g(1-\cos q)$$

beschrieben werden. Diese führt auf die Bewegungsgleichungen

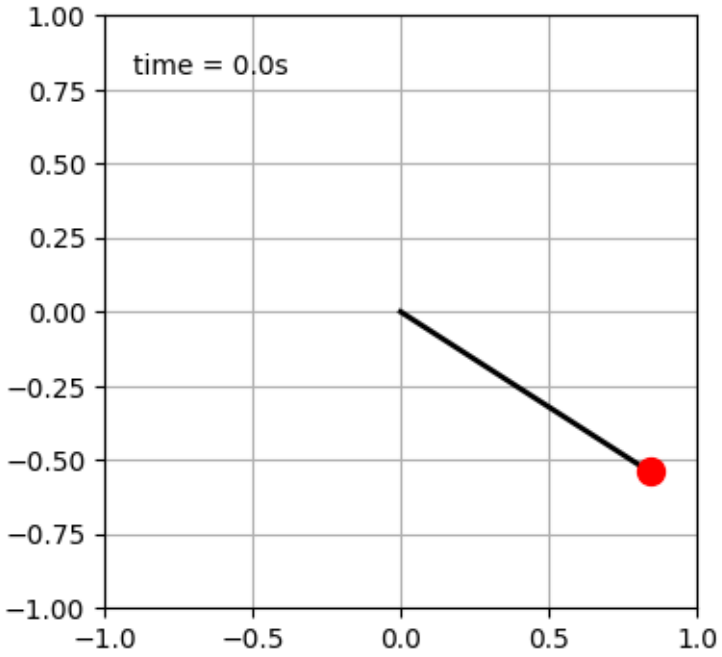
$$\dot{p} = -m\ell g \sin q, \quad \dot{q} = \frac{1}{m\ell^2}p.$$

[37]:
```
## parameters
g = 9.81                    # grav. acc.
ell = 1                     # length
m = 1.0                     # mass
## initial state
```

```
q0 = 1.0                    # initial angle (in radians)
p0 = 0.0                    # initial velocity
t0 = 0                      # intial time
y0 = np.array([p0, q0])     # intial state
# time span
tspan = (0.0,2*np.pi)
```

Berechnen Sie aus dem Anfangswinkel `q_0` die kartesischen Koordinaten x_1, y_1. Anschließend können Sie den vorhandenen code benutzen, um den Anfangszustand des Pendels darzustellen.

[38]:
```
x1 = ell * np.sin(q0)
y1 = -ell * np.cos(q0)

fig = plt.figure(figsize=(5, 4))
ax = fig.add_subplot(autoscale_on=False, xlim=(-ell, ell), ylim=(-ell, 1.))
ax.set_aspect('equal')
ax.grid()

plt.plot([0, x1], [0, y1], 'k-', lw=2)            # rod
plt.plot(x1, y1, 'ro', ms=10 * m)                # mass
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
time_text.set_text(time_template % (t0))
```



2.1 a) Zeitintegration mit Euler-Verfahren

Beginnen Sie damit, das mathematische Pendel mit dem expliziten Euler zu simulieren. Benutzen Sie dafür die folgende Vorlage:

[39]:
```
rhs_pendulum = lambda y: np.concat((-m * ell * g * np.sin(y[y.shape[0]//2:]),
                                    1 / (m * ell ** 2) * y[0:y.shape[0]//2]),
                                    axis=0)

def ForwardEulerIntegrate(y0, rhs, tspan, numTimeSteps=100):
    """ ForwardEulerIntegrate uses the forward Euler method to integrate the␣
 ↪differential
    equation y' = rhs(y) with initial state y(0) = y0

    y0 : initial state
    rhs : right-hand side of the differential equation
    tspan : time span (start_time,end_time)
    numTimeSteps : number of timesteps

    Returns:
    times : array of time points tn
    states : array of states yn over time
    """
    # Start by making arrays times and states
    # calculate tau and save the initial condition within states
    times = np.linspace(tspan[0], tspan[1], num=numTimeSteps+1)
    states = np.empty((y0.shape[0], numTimeSteps+1))
    tau = (tspan[1] - tspan[0]) / numTimeSteps
    states[:,0] = y0
    # Make a loop which increments a time step variable
    # In each step of the loop do a step with the explicit Euler method for␣
 ↪given rhs
    # Generate and save the calculated states in the array you defined earlier
    for i in range(0, numTimeSteps):
      y_n = states[:,i]
      y_new = y_n + tau * rhs_pendulum(y_n)
      states[:,i+1] = y_new
    # Finally give a return statement which returns the times and states
    return times, states
```

Wir können das Pendel über die Zeit hinweg visualisieren, indem wir den 'interact' decorator von 'ipywidgets' benutzen.

[40]:
```
from ipywidgets import interact
import ipywidgets as widgets

numTimeSteps = 1000
```

```
times_forward_angle, states_forward_angle = ForwardEulerIntegrate(y0,␣
 ↪rhs_pendulum, tspan,numTimeSteps = numTimeSteps)



@interact(n=widgets.IntSlider(min=0, max=numTimeSteps, step=1, value=0))
def visualize_pendulum(n):
    x = ell * np.sin(states_forward_angle[1,n])
    y = -ell * np.cos(states_forward_angle[1,n])

    fig, ax = plt.subplots(figsize=(5, 4))
    ax.set_xlim(-ell - 0.2, ell + 0.2)
    ax.set_ylim(-ell - 0.2, 1.0)
    ax.set_aspect('equal')
    ax.grid(True)
    # Trace of past positions
    history_x = ell * np.sin(states_forward_angle[1,:n])
    history_y = -ell * np.cos(states_forward_angle[1,:n])
    ax.plot(history_x, history_y, 'r-', lw=0.5, alpha=0.6)
    # Draw rod
    ax.plot([0, x], [0, y], 'k-', lw=2)
    # Draw bob
    ax.plot(x, y, 'ro', ms=10 * m)
    # Time label
    time_template = 'time = %.2fs'
    ax.text(0.05, 0.9, time_template % times_forward_angle[n], transform=ax.
 ↪transAxes)
    plt.show()
```

```
interactive(children=(IntSlider(value=0, description='n', max=1000), Output()),␣
 ↪_dom_classes=('widget-interact…
```

Was kann man beobachten? Was passiert, wenn Sie nur 100 Zeitschritte machen oder eine größere Endzeit T wählen?

<– Platz für Ihre Antwort –>

Das Pendel schwingt immer höher, die Gesamtenergie nimmt also zu, Macht man nur 100 Zeitschritte oder wählt meine eine größere Endzeit `T`, so nimmt die Energie schneller zu und das Pendel überschlägt sich

Als nächstes wollen wir das symplektische Euler-Verfahren benutzen. Für $H(p,q) = T(p) + U(q)$ wählen wir die Variante

$$p_{n+1} = p_n - \tau \nabla_q H(p_{n+1}, q_n) = p_n - \tau \nabla U(q_n)$$
$$q_{n+1} = q_n + \tau \nabla_p H(p_{n+1}, q_n) = q_n + \tau \nabla H(p_{n+1}).$$

Warum wäre die andere Variante für unser Beispiel ungeschickt?

Wie in der Vorlesung besprochen sind beide Methoden sinnvoll, da H separierbar

[41]:
```
def SymplecticEulerIntegrate(y0, dTdp, dUdq, tspan, numTimeSteps=100):
    """ SymplecticEulerIntegrate uses the symplectic Euler method to integrate
    a separable Hamiltonian system with H(p,q) = T(p) + U(q)

    y0 : initial state [p0, q0]
    dTdp : derivative of kinetic energy T with respect to p
    dUdq : derivative of potential energy U with respect to q
    tspan : time span (start_time, end_time)
    numTimeSteps : number of timesteps

    Returns:
    times : array of time points tn
    states : array of states [p_n, q_n] over time
    """
    # Proceed as in ForwardEulerIntegrate(...)
    times = np.linspace(tspan[0], tspan[1], num=numTimeSteps+1)
    d = y0.shape[0] // 2
    states = np.empty((2*d, numTimeSteps+1))
    states[:,0] = y0
    tau = (tspan[1] - tspan[0]) / numTimeSteps

    for i in range(0, numTimeSteps):
      pn = states[0:d,i]
      qn = states[d:,i]
      pnew = pn - tau * dUdq(qn)
      qnew = qn + tau * dTdp(pnew)
      states[0:d,i+1] = pnew
      states[d:,i+1] = qnew

    return times, states

def SymplecticEulerIntegrateAdjoint(y0, dTdp, dUdq, tspan, numTimeSteps=100):
    """ SymplecticEulerIntegrateAdjoint uses the symplectic Euler method to␣
 ↪integrate
    a separable Hamiltonian system with H(p,q) = T(p) + U(q)

    qn, pn --> qn+1 --> pn+1

    y0 : initial state [p0, q0]
    dTdp : derivative of kinetic energy T with respect to p
    dUdq : derivative of potential energy U with respect to q
    tspan : time span (start_time, end_time)
    numTimeSteps : number of timesteps

    Returns:
    times : array of time points tn
    states : array of states [p_n, q_n] over time
```

```
    """
    # Proceed as in ForwardEulerIntegrate(...)
    times = np.linspace(tspan[0], tspan[1], num=numTimeSteps+1)
    d = y0.shape[0] // 2
    states = np.empty((2*d, numTimeSteps+1))
    states[:,0] = y0
    tau = (tspan[1] - tspan[0]) / numTimeSteps

    for i in range(0, numTimeSteps):
      pn = states[0:d,i]
      qn = states[d:,i]
      qnew = qn + tau * dTdp(pn)
      pnew = pn - tau * dUdq(qnew)
      states[0:d,i+1] = pnew
      states[d:,i+1] = qnew

    return times, states
```

Wie zuvor für den expliziten Euler, können wir auch die mithilfe des symplektischen Eulers berechnete Lösung visualisieren. Was fällt im Vergleich auf?

<– Platz für Ihre Antwort –>

Das Pendel hat immer ungefähr die selbe Gesamtenergie, also den selben Maximalausschlag, insbesondere für längere Zeiten

[42]:
```
# Visualize also the simulation running the symplectic Euler method

from ipywidgets import interact
import ipywidgets as widgets

dTdp = lambda p: 1 / (m * ell ** 2) * p
dUdq = lambda q: m * ell * g * np.sin(q)

numTimeSteps = 1000
times_sym_angle, states_sym_angle = SymplecticEulerIntegrate(y0, dTdp, dUdq,␣
 ↪tspan, numTimeSteps = numTimeSteps)



@interact(n=widgets.IntSlider(min=0, max=numTimeSteps, step=1, value=0))
def visualize_pendulum(n):
    x = ell * np.sin(states_sym_angle[1,n])
    y = -ell * np.cos(states_sym_angle[1,n])

    fig, ax = plt.subplots(figsize=(5, 4))
    ax.set_xlim(-ell - 0.2, ell + 0.2)
    ax.set_ylim(-ell - 0.2, 1.0)
```

```
    ax.set_aspect('equal')
    ax.grid(True)
    # Trace of past positions
    history_x = ell * np.sin(states_sym_angle[1,:n])
    history_y = -ell * np.cos(states_sym_angle[1,:n])
    ax.plot(history_x, history_y, 'r-', lw=0.5, alpha=0.6)
    # Draw rod
    ax.plot([0, x], [0, y], 'k-', lw=2)
    # Draw bob
    ax.plot(x, y, 'ro', ms=10 * m)
    # Time label
    time_template = 'time = %.2fs'
    ax.text(0.05, 0.9, time_template % times_sym_angle[n], transform=ax.
 ↪transAxes)
    plt.show()
```

```
interactive(children=(IntSlider(value=0, description='n', max=1000), Output()),␣
 ↪_dom_classes=('widget-interact…
```

Zuletzt verwende wir den impliziten Euler. Die zusätzliche Hauptschwierigkeit liegt dabei in der Lösung des auftretenden nicht-linearen Gleichungssystems. Hierfür verwenden wir die Funktion `scipy-optimize.fsolve`.

[43]:
```
from scipy.optimize import fsolve

def BackwardEulerIntegrate(y0, rhs, tspan, numTimeSteps=100):
    """
    Backward Euler method to integrate y' = rhs(y) with initial state y0.

    Parameters:
    y0 : initial state (as numpy array)
    rhs : function rhs(y) returning dy/dt
    tspan : tuple (start_time, end_time)
    numTimeSteps : number of time steps

    Returns:
    times : array of time points
    states : 2D array of shape (len(y0), len(times)) containing the states at␣
 ↪each time
    """
    t0, T = tspan
    times = np.linspace(t0, T, numTimeSteps + 1)
    tau = (T - t0) / numTimeSteps

    y = y0.copy()
    states = np.zeros((len(y0), len(times)))
    states[:, 0] = y
```

```
    def implicit_system(y_next, y_current):
        return y_current - y_next + tau * rhs(y_next)

    for i in range(numTimeSteps):
        estimate = y
        y = fsolve(implicit_system, estimate, args=(y,))
        states[:, i + 1] = y

    return times, states
```

Setzen Sie sich mit der Dokumentation von `scipy.optimize.fsolve` auseinander. Wofür brauchen wir die Funktion `implicit_system` und was muss diese erfüllen? Warum ergibt es Sinn, `y0` als Schätzung zu übergeben?

<– Platz für Ihre Antwort –>

`fsolve` approximiert eine Lösung für `implicit_system(y_next, ...) = 0`, `implicit_system` muss einen Vektor als erstes Argument erwarten und einen Vektor zurückgeben ✓

Da `y1 - y0` $\in O(\tau)$ macht `y0` als Schätzung Sinn für `y1` ✓

[44]:
```
# Visualize also the simulation running the backward Euler method

from ipywidgets import interact
import ipywidgets as widgets

numTimeSteps = 1000
times_backward_angle, states_backward_angle = BackwardEulerIntegrate(y0,␣
 ↪rhs_pendulum, tspan, numTimeSteps = numTimeSteps)



@interact(n=widgets.IntSlider(min=0, max=numTimeSteps, step=1, value=0))
def visualize_pendulum(n):
    x = ell * np.sin(states_backward_angle[1,n])
    y = -ell * np.cos(states_backward_angle[1,n])

    fig, ax = plt.subplots(figsize=(5, 4))
    ax.set_xlim(-ell - 0.2, ell + 0.2)
    ax.set_ylim(-ell - 0.2, 1.0)
    ax.set_aspect('equal')
    ax.grid(True)
    # Trace of past positions
    history_x = ell * np.sin(states_backward_angle[1,:n])
    history_y = -ell * np.cos(states_backward_angle[1,:n])
    ax.plot(history_x, history_y, 'r-', lw=0.5, alpha=0.6)
    # Draw rod
    ax.plot([0, x], [0, y], 'k-', lw=2)
    # Draw bob
```

```
    ax.plot(x, y, 'ro', ms=10 * m)
    # Time label
    time_template = 'time = %.2fs'
    ax.text(0.05, 0.9, time_template % times_backward_angle[n], transform=ax.
↪transAxes)
    plt.show()
```

```
interactive(children=(IntSlider(value=0, description='n', max=1000), Output()),␣
↪_dom_classes=('widget-interact…
```

Was passiert im Falle des impliziten Eulers, wenn Sie die Endzeit erhöhen oder die Anzahl der Zeitschritte verringern?

<– Platz für Ihre Antwort –>

Die Gesamtenergie, am Maximalausschlag des Pendels zu erkennen, wird geringer, erhöhen wir die Endzeit oder verringern wir die Anzahl der Zeitschritte, so nimmt die Energie deutlicher oder schneller ab

2.2 b) Konvergenzanalyse und Erhaltungsgrößen

Als nächstes wollen wir für obige Verfahren das jeweilige Konvergenzverhalten für $\tau \to 0$ untersuchen. Dafür können wir entweder versuchen eine exakte Lösung zu konstruieren oder gegen eine Referenzlösung testen.

Die exakte Lösung des mathematischen Pendels ist im allgemeinen Fall kompliziert. Aus diesem Grund betrachten wir ein vereinfachtes Problem, indem wir die Kleinwinkelnäherung

$$\sin q \approx q$$

verwenden. Diese ist für geringe Auslenkungen hinreichend akkurat. Beispielsweise gilt

[45]:
```
np.sin(0.05) - 0.05
```

[45]:
```
np.float64(-2.083072932167196e-05)
```

Durch Substitution ergibt sich somit die lineare Differentialgleichung zweiter Ordnung

$$\ddot{q} + \frac{g}{\ell} q = 0 \,.$$

Leiten Sie unter der Annahme der Kleinwinkelnäherung eine exakte Lösung für das mathematische Pendel her.

<– Platz für Ihre Antwort –>

$$\begin{pmatrix} \dot{q} \\ \ddot{q} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ a & 0 \end{pmatrix} \begin{pmatrix} q \\ \dot{q} \end{pmatrix}$$

charakteristisches Polynom: $\lambda^2 - a = 0 \iff \lambda \pm \sqrt{a} = \pm\sqrt{-\frac{g}{l}} = \pm\sqrt{\frac{g}{l}}i$

$$E_{\sqrt{\frac{g}{l}}i} = \left[\begin{pmatrix} \sqrt{\frac{l}{g}}i \\ 1 \end{pmatrix} \right]$$

Lösungen: Fundamentalsystem

$$\left[\begin{pmatrix}\cos\sqrt{\frac{g}{l}}t\\ \sin\sqrt{\frac{g}{l}}t\end{pmatrix} \quad \begin{pmatrix}\sin\sqrt{\frac{g}{l}}t\\ \cos\sqrt{\frac{g}{l}}t\end{pmatrix}\right]$$

$$q(t) = q(0)\cos\sqrt{\frac{g}{l}}t + \dot{q}(0)\sin\sqrt{\frac{g}{l}}t$$

Benutzen Sie die folgende Vorlage, um die exakte Lösung anzulegen. Achten Sie darauf, dass Ihre Implementierung Vektorisierung erlaubt.

[46]:
```
m = 1.0
g_small_angle = 9.81

def exaktSolutionSmallAnglePendulum(q0=0.05):
    ### Define the exact solution for a small angle
    sol = lambda t: q0 * np.cos(np.sqrt(g_small_angle / ell) * t) # assuming␣
 ↪q'(0) = 0
    return sol
sol = exaktSolutionSmallAnglePendulum()
### make sure you can evaluate your solution also vectorized
sol(np.array([0.0,0.5,1.0]))
```

[46]:
```
array([ 0.05      ,  0.00023752, -0.04999774])
```

Wieder können wir die Lösung visualisieren, indem wir den Winkel in geeignete Koordinaten überführen.

[47]:
```
# Visualize also the simulation running exaktSolutionSmallAnglePendulum(0.05)

from ipywidgets import interact
import ipywidgets as widgets

numTimeSteps = 1000
sol_small_angle = exaktSolutionSmallAnglePendulum(0.05)


vis_eval = np.linspace(tspan[0], tspan[1], numTimeSteps+1)

@interact(n=widgets.IntSlider(min=0, max=numTimeSteps, step=1, value=0))
def visualize_pendulum(n):
    q = sol_small_angle(vis_eval)
    x = ell * np.sin(q[n])
    y = -ell * np.cos(q[n])

    fig, ax = plt.subplots(figsize=(5, 4))
    ax.set_xlim(-ell - 0.2, ell + 0.2)
    ax.set_ylim(-ell - 0.2, 1.0)
    ax.set_aspect('equal')
    ax.grid(True)
```

```
    # Trace of past positions
    history_x = ell * np.sin(q[:n])
    history_y = -ell * np.cos(q[:n])
    ax.plot(history_x, history_y, 'r-', lw=0.5, alpha=0.6)
    # Draw rod
    ax.plot([0, x], [0, y], 'k-', lw=2)
    # Draw bob
    ax.plot(x, y, 'ro', ms=10 * m)
    # Time label
    time_template = 'time = %.2fs'
    ax.text(0.05, 0.9, time_template % vis_eval[n], transform=ax.transAxes)
    plt.show()
```

```
interactive(children=(IntSlider(value=0, description='n', max=1000), Output()),␣
↪_dom_classes=('widget-interact…
```

Erstellen Sie für den expliziten Euler, den symplektischen Euler und den impliziten Euler Konvergenzplots für $ $\to 0$. Messen Sie als Fehler den maximalen Fehler des Winkels q über alle Stützstellen $t_n \in [0, 2\pi]$. Verwenden Sie mindestens 100 und maximal 5000 Zeitschritte.

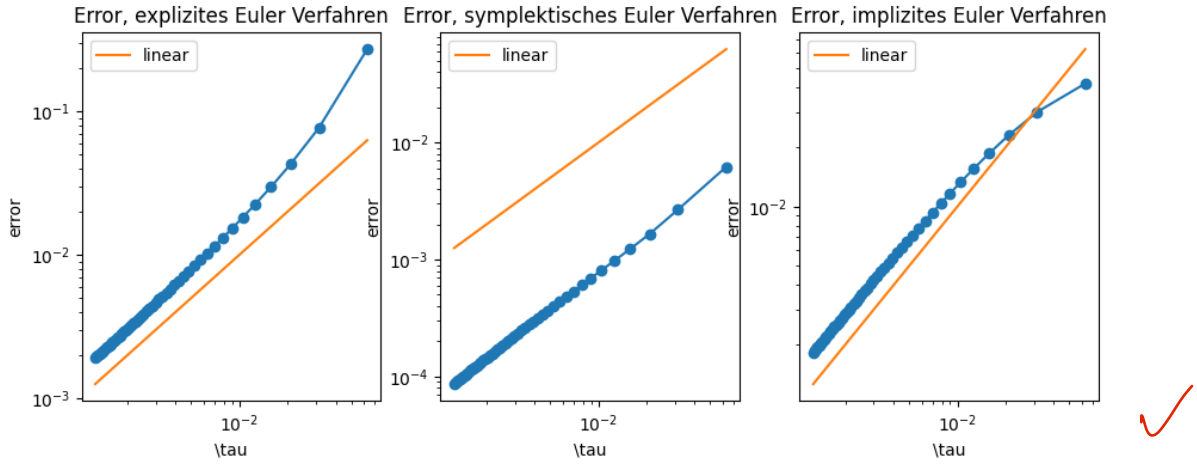
[47]:

[48]:
```
# Initial data
q0 = 0.05
p0 = 0.0

tau_f = lambda numTimeStep, tspan: (tspan[1] - tspan[0]) / numTimeStep
max_abs = lambda ref_q, q: np.max(np.abs(ref_q - q))
y0 = np.array([p0, q0])

def calculate_errors(ref_integrator, integrators, numTimeStepStep, tspan,␣
↪error_function=max_abs, y0=y0, **kwargs):
  """
  Calculates the errors of the integrators compared to the reference integrator

  Parameters:
  ref_integrator: reference integrator (y0, t_eval, tspan ...) --> array of qs
  integrators: dict of (integrator name: integrator)
  numTimeStepStep: granularity of numTimeStep range
  tspan:
  error_function: ref_q, q --> error
  kwargs: any additional parameters for the reference integrator

  Returns:
  dict of (integrator name: error array), numTimeStep range
  """

  minNumTimeStep = 100
```

```
  maxNumTimeStep = 5000
  numTimeStepSpan = range(maxNumTimeStep, minNumTimeStep-1, -numTimeStepStep)
  integrator_errors = {name: np.empty(((maxNumTimeStep - minNumTimeStep) //
                                       numTimeStepStep + 1)) for (name, _)
                                       in integrators.items()}

  for i, numTimeStep in enumerate(numTimeStepSpan):
    # reference solution
    t_eval = np.linspace(tspan[0], tspan[1], num=numTimeStep+1)
    ref_q = ref_integrator(y0, t_eval, tspan, **kwargs)

    for name, integrator in integrators.items():
      q = integrator(y0, t_eval, tspan)
      integrator_errors[name][i] = error_function(ref_q, q)

  return integrator_errors, numTimeStepSpan

def plot_errors(integrator_errors, numTimeStepSpan, combine=False,
                ref_f=(lambda x: x), ref_label="linear", axes=False,
                xscale="linear", yscale="linear", title="Error"):
  """
  Visualizes the errors

  Parameters:
  integrator_errors: dict of (integrator name, array of errors)
  numTimeStepSpan: numTimeStep range
  combine: whether to visualize in one image
  xscale: scale of the x axis
  yscale: scale of the y axis

  Returns:
  if axs = True
  axes
  if acs = False
  void
  """

  tau_l = [tau_f(numTimeStep, tspan) for numTimeStep in numTimeStepSpan]

  if combine or len(integrator_errors) == 1:
    fig, axs = plt.subplots(1, 1, figsize=(4, 4))
    axs.set_xlabel("\\tau")
    axs.set_ylabel("error")
    axs.set_xscale(xscale)
    axs.set_title(title)
    axs.set_yscale(yscale)
    for name, errors in integrator_errors.items():
```

```
      axs.plot(tau_l, errors, marker="o", linestyle="-", label=name)
    axs.plot(tau_l, ref_f(tau_l), marker="", linestyle="-", label=ref_label)
    axs.legend()
  else:
    fig, axs = plt.subplots(1, len(integrator_errors), figsize=(11, 4))
    for ax, (name, errors) in zip(axs, integrator_errors.items()):
      ax.set_title(title + ", " + name)
      ax.set_xlabel("\\tau")
      ax.set_xscale(xscale)
      ax.set_yscale(yscale)
      ax.set_ylabel("error")
      ax.plot(tau_l, errors, marker="o", linestyle="-")
      ax.plot(tau_l, ref_f(tau_l), marker="", linestyle="-", label=ref_label)
      ax.legend()
  if axes:
    plt.close(fig)
    return fig.axes
  else:
    plt.show()

def plot_axs(axs, title, nrow, ncol):
  """
  Visualizes the axes as one figure

  Parameters:
  axs: one dimensional list of axes
  title: title of the figure
  nrow: number of subplots rows
  ncol: number of subplots cols

  Returns:

  """

  fig, new_axs = plt.subplots(nrow, ncol, figsize=(ncol * 4, nrow * 4))
  fig.suptitle(title)
  for ax, new_ax in zip(axs, new_axs.flatten()):
    for line in ax.get_lines():
      new_ax.plot(line.get_xdata(), line.get_ydata(), label=line.get_label())
    new_ax.set_title(ax.get_title())
    new_ax.set_xlabel(ax.get_xlabel())
    new_ax.set_ylabel(ax.get_ylabel())
    new_ax.set_xscale(ax.get_xscale())
    new_ax.set_yscale(ax.get_yscale())
    if legend := ax.get_legend():
      handles, labels = ax.get_legend_handles_labels()
      new_ax.legend(handles=handles, labels=labels)
```

```
  plt.plot()

exakt_ref = lambda y0, t_eval, tspan:␣
 ↪exaktSolutionSmallAnglePendulum(y0[1])(t_eval)
exp_euler = lambda y0, t_eval, tspan: ForwardEulerIntegrate(y0, rhs_pendulum,␣
 ↪tspan, t_eval.shape[0]-1)[1][1,:]
sym_euler = lambda y0, t_eval, tspan: SymplecticEulerIntegrate(y0, dTdp, dUdq,␣
 ↪tspan, t_eval.shape[0]-1)[1][1,:]
imp_euler = lambda y0, t_eval, tspan: BackwardEulerIntegrate(y0, rhs_pendulum,␣
 ↪tspan, t_eval.shape[0]-1)[1][1,:]
integrators = {"explizites Euler Verfahren": exp_euler, "symplektisches Euler␣
 ↪Verfahren": sym_euler, "implizites Euler Verfahren": imp_euler}
integrator_errors, numTimeStepSpan = calculate_errors(exakt_ref, integrators,␣
 ↪100, tspan)
plot_errors(integrator_errors, numTimeStepSpan, xscale="log", yscale="log")
```



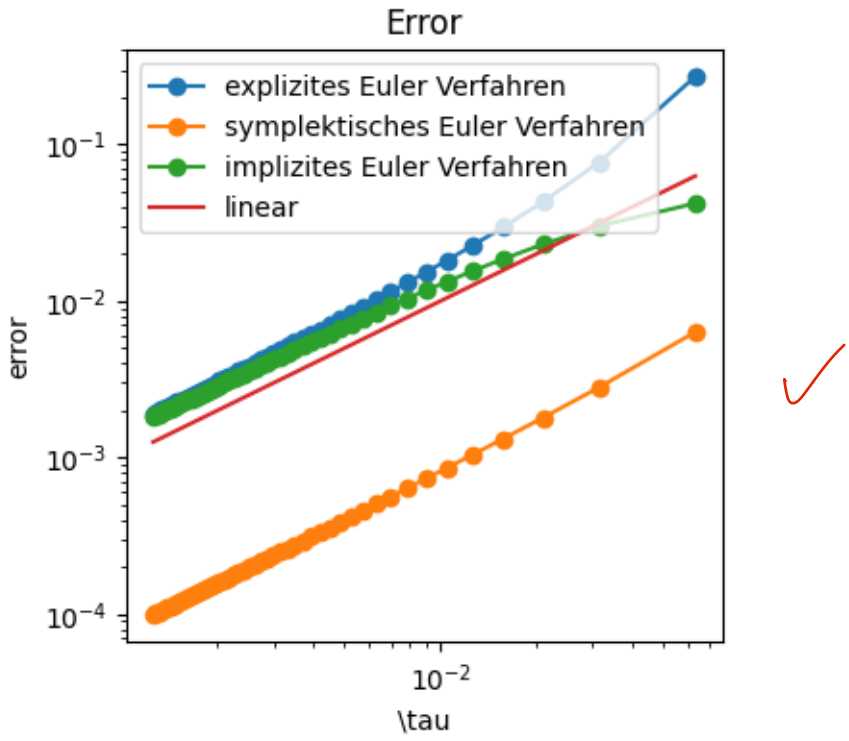
da log log Scale $\log Ch^e = \log C + e \cdot \log h$ folgt die Ordnung des Verfahrens aus der Steigung

Wir wollen die Verfahren allerdings nicht nur für kleine Winkel q_0 untersuchen. Mit dem folgenden Code können Sie eine sogenannte Referenzlösung berechnen. Dafür kann auf die `scipy` Implementierung von `solve_ivp` zugegriffen werden. Nutzen Sie den Parameter `t_eval` um die approximierte Lösung an denselben Zeitschritten auszuwerten wie bei den Euler-Verfahren.

[49]:
```
import scipy.integrate

rhs = lambda _, y: rhs_pendulum(y)

def referenceSolutionRK45(y0, rhs, tspan, times, tol=1e-8):
    sol = scipy.integrate.solve_ivp(rhs, tspan, y0, t_eval=times, atol=tol,␣
 ↪rtol=tol, method='RK45')
    return sol.y
```

Nutzen Sie diese Referenzlösung, um Konvergenzplots zu erzeugen. Nutzen Sie die Startwerte $q_0 = 1$ und $p_0 = 0$. Plotten Sie diesmal alle Fehlerlinien in ein einzelnes Schaubild.

[50]:
```
q0 = 1
p0 = 0

ref_sol = lambda y0, times, tspan, rhs, tol: referenceSolutionRK45(y0, rhs,␣
 ↪tspan, times, tol)[1,:]

integrator_errors, numTimeStepSpan = calculate_errors(ref_sol, integrators,␣
 ↪100, tspan=tspan, rhs=rhs, tol=1e-8)
plot_errors(integrator_errors, numTimeStepSpan, combine=True, xscale="log",␣
 ↪yscale="log")
```



Wie wir also sehen, konvergieren alle drei Verfahren mit Ordnung 1. Allerdings reicht es in der Praxis manchmal nicht, sich ausschließlich mit der Konvergenz des Fehlers zu beschäftigen. Oft sind wir zum Beispiel daran interessiert, die Erhaltung gewisser Größen auch im numerischen Verfahren sicher zu stellen. Im Falle des Pendels könnten wir zum Beispiel daran interessiert sein, die Gesamtenergie zu erhalten.

Die physikalische Gesamtenergie ist durch

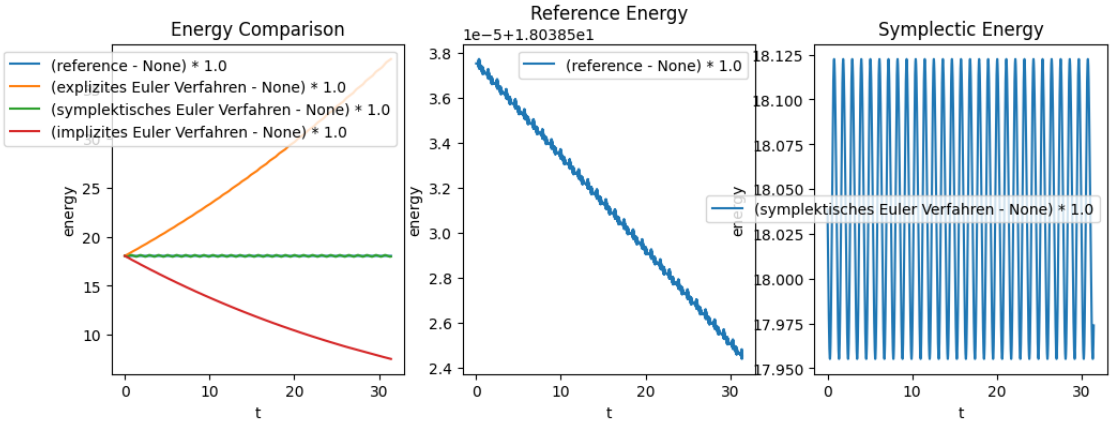
$$H(p,q) = \frac{1}{2m\ell^2}p^2 + m\ell g(1-\cos q)$$

gegeben.

Plotten Sie die Energie der Approximationen über das Zeitinterval $[0, 10\pi]$ mit 10000 Zeitschritten für alle drei Euler-Verfahren. Wählen Sie dabei als Masse `m=4.0`. Plotten Sie auch die Energie der Referenzlösung.

[51]:
```
# Initial data
q0 = 1.0
p0 = 0.0
y0 = np.array([p0, q0])
T = 10 * np.pi
N = 10000
tspan = (0.0, T)
times = np.linspace(*tspan, N + 1)
# Physical parameters
g = 9.81
ell = 1.0
m = 4.0

energy = lambda y: (1 / (2 * m * ell ** 2)) * y[0,...] ** 2 + m * ell * g * (1␣
↪- np.cos(y[1,...]))


def plot_energy(energy_f, integrators, t_eval, y0, title, tspan, combine=False,␣
↪axes=False, c=None, ref=None):
  """
  Visualizes the energy over time

  Parameters:
  integrators: dict of (integrator name: integrator)
  t_eval: when to approximate
  y0: initial value
  title: title of image
  combine: whether to visualize all energies in one image
  c: scalar multiplicators for integrator energies
  ref: name of reference integrator

  Returns:

  """

  if c == None or len(c) != len(integrators):
    c = [1.0] * len(integrators)


  integrator_energies = {name: energy_f(integrator(y0, t_eval, tspan)) for␣
↪name, integrator in integrators.items()}
  ref_energy = 0
  if ref != None and ref in integrator_energies:
```

```
    ref_energy = integrator_energies[ref]

  if combine:
    fig, axs = plt.subplots(1, 1, figsize=(4, 4))
    axs.set_title(title)
    axs.set_xlabel("t")
    axs.set_ylabel("energy")
    for i, (name, integrator_energy) in enumerate(integrator_energies.items()):
      axs.plot(t_eval, c[i] * (integrator_energy - ref_energy), marker="",␣
 ↪linestyle="-", label="(" + name + f" - {ref}) * {c[i]}")
    axs.legend()
  else:
    fig, axs = plt.subplots(1, len(integrator_energies))
    for i, ax, (name, energy) in enumerate(zip(axs, integrator_energies.
 ↪items())):
      ax.set_title(f"Energy ({name})")
      ax.set_xlabel("t")
      ax.set_ylabel(f"(energy - {ref}) * ", c[i])
      ax.plot(t_eval, c[i] * (energy - ref_energy), marker="", linestyle="")
  if axes:
    plt.close(fig)
    return fig.axes
  else:
    plt.show()

ref_sol_all = lambda y0, t_eval, tspan: referenceSolutionRK45(y0, rhs, tspan,␣
 ↪t_eval, tol=1e-8)
exp_euler_all = lambda y0, t_eval, tspan: ForwardEulerIntegrate(y0,␣
 ↪rhs_pendulum, tspan, t_eval.shape[0]-1)[1]
sym_euler_all = lambda y0, t_eval, tspan: SymplecticEulerIntegrate(y0, dTdp,␣
 ↪dUdq, tspan, t_eval.shape[0]-1)[1]
imp_euler_all = lambda y0, t_eval, tspan: BackwardEulerIntegrate(y0,␣
 ↪rhs_pendulum, tspan, t_eval.shape[0]-1)[1]
integrators_all = {"reference": ref_sol_all, "explizites Euler Verfahren":␣
 ↪exp_euler_all, "symplektisches Euler Verfahren": sym_euler_all, "implizites␣
 ↪Euler Verfahren": imp_euler_all}
comparison_axs = plot_energy(energy, integrators_all, times, y0, "Energy␣
 ↪Comparison", tspan, combine=True, axes=True)
reference_ax = plot_energy(energy, {"reference": ref_sol_all}, times, y0,␣
 ↪"Reference Energy", tspan, combine=True, axes=True)
sym_ax = plot_energy(energy, {"symplektisches Euler Verfahren": sym_euler_all},␣
 ↪times, y0, "Symplectic Energy", tspan, combine=True, axes=True)
plot_axs(comparison_axs + reference_ax + sym_ax, "", 1, 3)
```



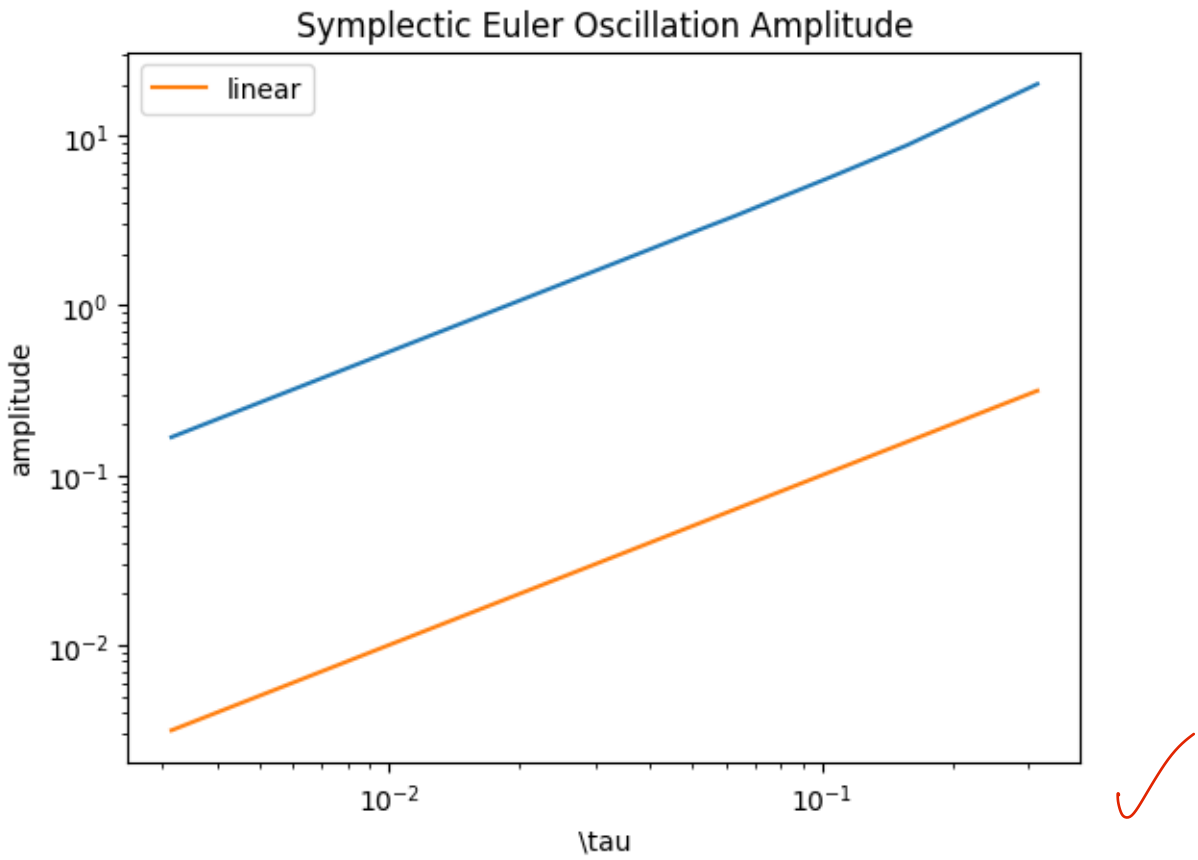
Wie unterscheiden sich die Verfahren? Was sehen Sie, wenn Sie nur die Referenzlösung darstellen? Was, wenn Sie nur den symplektischen Euler betrachten?

<– Platz für Ihre Antwort –>

Wie in dem Beispiel der Vorlesung wird die Energie der Lösung des expliziten Euler Verfahrens höher, die Energie des impliziten Euler Verfahrens geringer und die Energie des symplektischen Euler Verfahrens oszilliert im Bereich der Energie der Referenz,

die Energie der Referenz wird geringfügig geringer, da es eine Approximation und nicht die exakte Lösung

[52]:
```
minTimeStep = 100
maxTimeStep = 10000
a = 100
numTimeStepSpan = range(maxTimeStep, minTimeStep-1, -a)

amp = np.empty(((maxTimeStep - minTimeStep) // a + 1))
for i, N in enumerate(numTimeStepSpan):
  sym_energy = energy(SymplecticEulerIntegrate(y0, dTdp, dUdq, tspan, N)[1])
  amp[i] = np.max(sym_energy) - np.min(sym_energy)

tau_l = [tau_f(numTimeStep, tspan) for numTimeStep in numTimeStepSpan]
plt.plot(tau_l, amp, marker="", linestyle="-")
plt.plot(tau_l, tau_l, marker="", linestyle="-", label="linear")
plt.title("Symplectic Euler Oscillation Amplitude")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("\\tau")
plt.ylabel("amplitude")
plt.legend()
plt.show()

# symplektischer Euler die Oszillation in O(tau)
```



also die Oszillation des symplektischen Euler Verfahrens in $O(\tau)$ für Zeiten $[0, 10\pi]$ +0,5

3 Aufgabe 2 - Symplektische Verfahren höherer Ordnung 3/3

3.1 a) Ein Verfahren zweiter Ordnung

Wir betrachten in dieser Aufgabe wieder nur den Fall einer separablen Hamilton-Funktion $H(p, q) = T(p) + U(q)$. Das Leapfrog- (oder auch Störmer-Verlet-) Verfahren

$$\begin{aligned} p_{n+1/2} &= p_n - \frac{\tau}{2}\nabla U(q_n) \\ q_{n+1} &= q_n + \tau \nabla T(p_{n+1/2}) \\ p_{n+1} &= p_{n+1/2} - \frac{\tau}{2}\nabla U(q_{n+1}) \end{aligned}$$

ist ein Integrator zweiter Ordnung, siehe z.B. Gleichung (2.10) in diesem Acta Numerica Artikel.

Implementieren Sie dieses Verfahren und untersuchen Sie es auf Konvergenz und Energieerhaltung.

[53]:
```
def LeapfrogIntegrate(y0, dTdp, dUdq, tspan, numTimeSteps):
    """
    Leapfrog (Störmer-Verlet) integrator for separable Hamiltonian systems.
```

```
    Parameters:
    y0 : Initial state as array [p0, q0]
    dTdp : Derivative of the kinetic energy with respect to momentum
    dUdq : Derivative of the potential energy with respect to position
    tspan : Tuple (t0, T) for initial and final times
    numTimeSteps : Number of time steps

    Returns:
    times : Time points as array of shape (numTimeSteps + 1,)
    states : Array of shape (2, numTimeSteps + 1) containing momenta and␣
↪positions over time
    """

    times = np.linspace(tspan[0], tspan[1], numTimeSteps+1)
    tau = tau_f(numTimeSteps, tspan)
    states = np.empty((2, numTimeSteps+1))
    states[:, 0] = y0

    for i in range(0, numTimeSteps):
      pn = states[0, i]
      qn = states[1, i]

      p_half = pn - 0.5 * tau * dUdq(qn)
      q_new = qn + tau * dTdp(p_half)
      p_new = p_half - 0.5 * tau * dUdq(q_new)

      states[0, i+1] = p_new
      states[1, i+1] = q_new

    return times, states
```

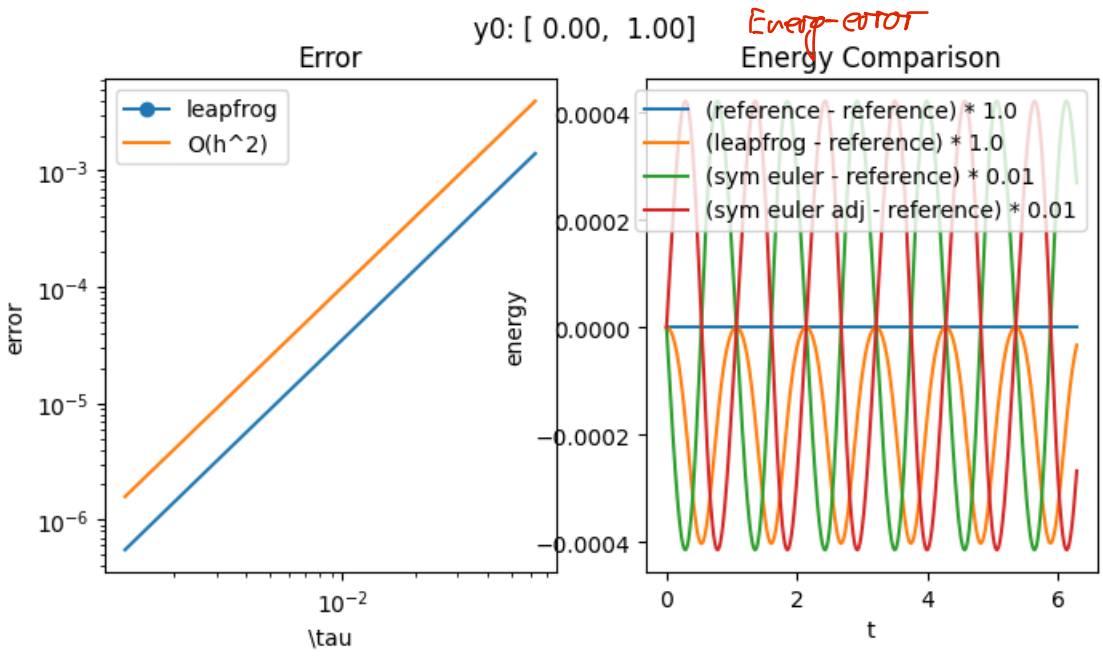
Untersuchen Sie das Verfahren nun auf Konvergenz (gegen die Referenzlösung) und auf Energieerhaltung. Erklären Sie anschließend, was sie sehen. Variieren Sie außerdem die Anfangswerte.

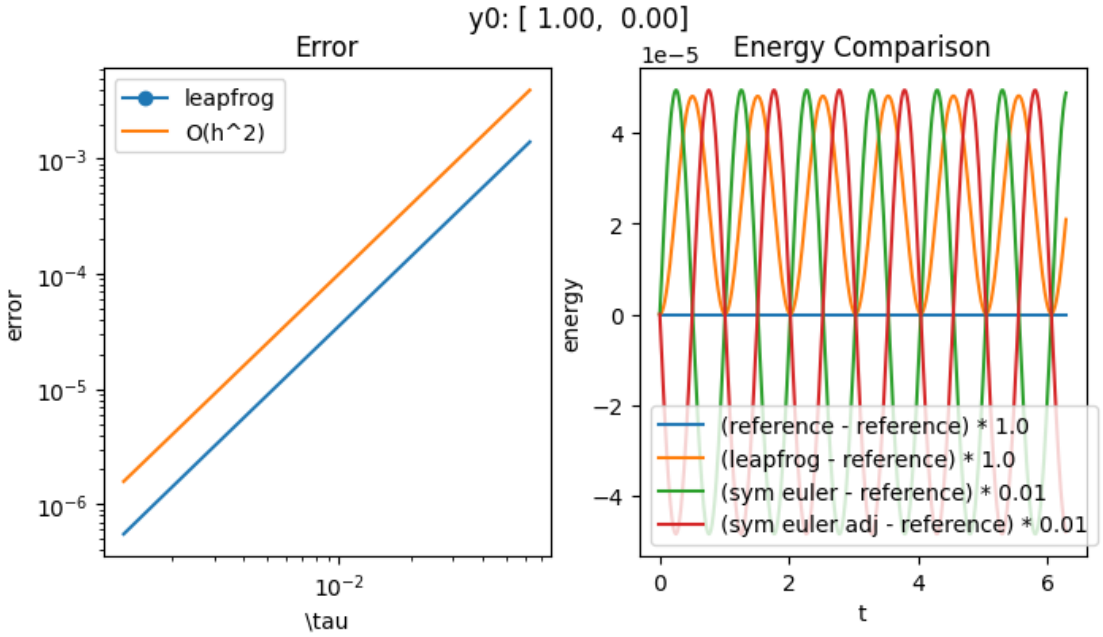
[54]:
```
# Parameters
q0 = 1.0
p0 = 0.0
y01 = np.array([p0, q0])
t0 = 0.0
T = 2 * np.pi
N = 1000
tspan = (t0, T)
m = 1.0
ell = 1.0
g = 9.81
```

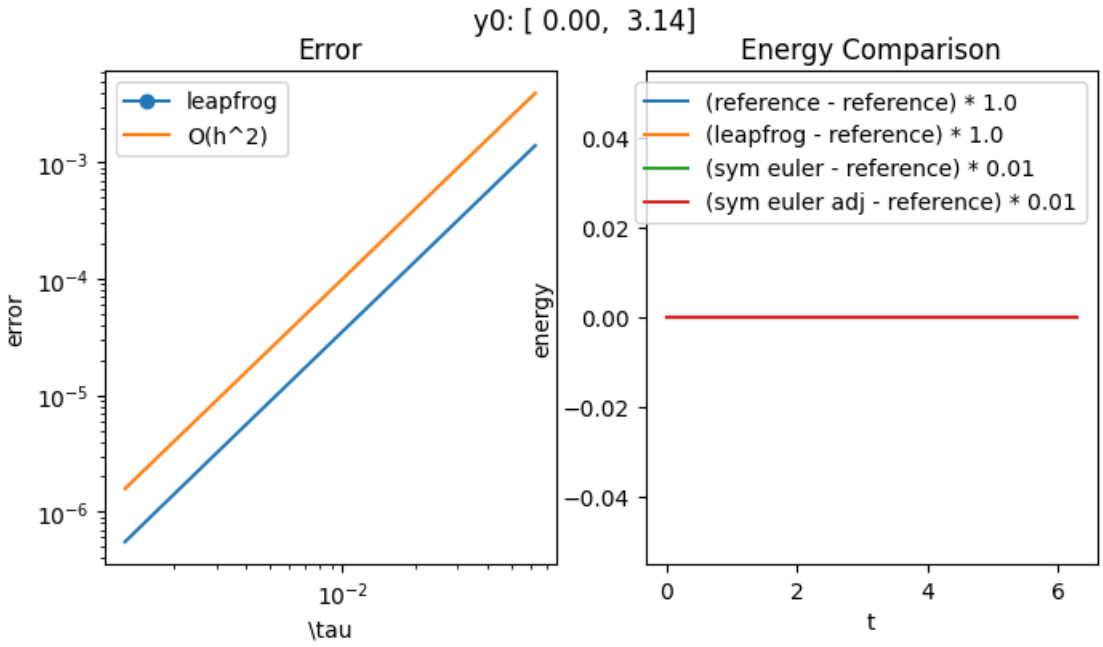
```
# quadratic reference
quadratic = lambda x: np.pow(x, 2)

y0l = [y01, np.array([1.0, 0.0]), np.array([0, np.pi]), np.array([0, np.pi /␣
 ↪2]), np.array([0.0, 0.0]), np.array([3.0, 0.1]), np.array([3.0, 1.0]), np.
 ↪array([0, - np.pi / 2])]

sym_euler_adj_all = lambda y0, t_eval, tspan:␣
 ↪SymplecticEulerIntegrateAdjoint(y0, dTdp, dUdq, tspan, t_eval.shape[0]-1)[1]

leapfrog = lambda y0, t_eval, tspan: LeapfrogIntegrate(y0, dTdp, dUdq, tspan,␣
 ↪t_eval.shape[0]-1)[1][1, :]
ref_sol_all = lambda y0, t_eval, tspan: referenceSolutionRK45(y0, rhs, tspan,␣
 ↪t_eval, tol=1e-8)
for y0 in y0l:
  integrators = {"leapfrog": leapfrog}
  integrator_errors, numTimeStepSpan = calculate_errors(ref_sol, integrators,␣
 ↪100, tspan=tspan, rhs=rhs, tol=1e-8)
  error_ax = plot_errors(integrator_errors, numTimeStepSpan, ref_f=quadratic,␣
 ↪ref_label="O(h^2)", combine=True, xscale="log", yscale="log",␣
 ↪title=f"Error", axes=True)

  leapfrog_all = lambda y0, t_eval, tspan: LeapfrogIntegrate(y0, dTdp, dUdq,␣
 ↪tspan, t_eval.shape[0]-1)[1]
  integrators_all = {"reference": ref_sol_all, "leapfrog": leapfrog_all, "sym␣
 ↪euler": sym_euler_all, "sym euler adj": sym_euler_adj_all}
  energy_ax = plot_energy(energy, integrators_all, np.linspace(t0, T, N+1), y0,␣
 ↪f"Energy Comparison", tspan, combine=True, axes=True, c=[1.0, 1.0, 0.01, 0.
 ↪01], ref="reference")

  plot_axs(error_ax + energy_ax, f"y0: [{y0[0]: .2f}, {y0[1]: .2f}]", 1, 2)



"""import seaborn as sns

methods = [leapfrog_all, sym_euler_all, sym_euler_adj_all]
h_l = []
x_min = -3.0
x_max = 3.0
y_min = -np.pi
y_max = np.pi
for method in methods:
  energy = lambda y: (1 / (2 * m * ell ** 2)) * y[0,...] ** 2 + m * ell * g *␣
 ↪(1 - np.cos(y[1,...]))
  X = np.arange(x_min, x_max, 0.3).tolist()
```

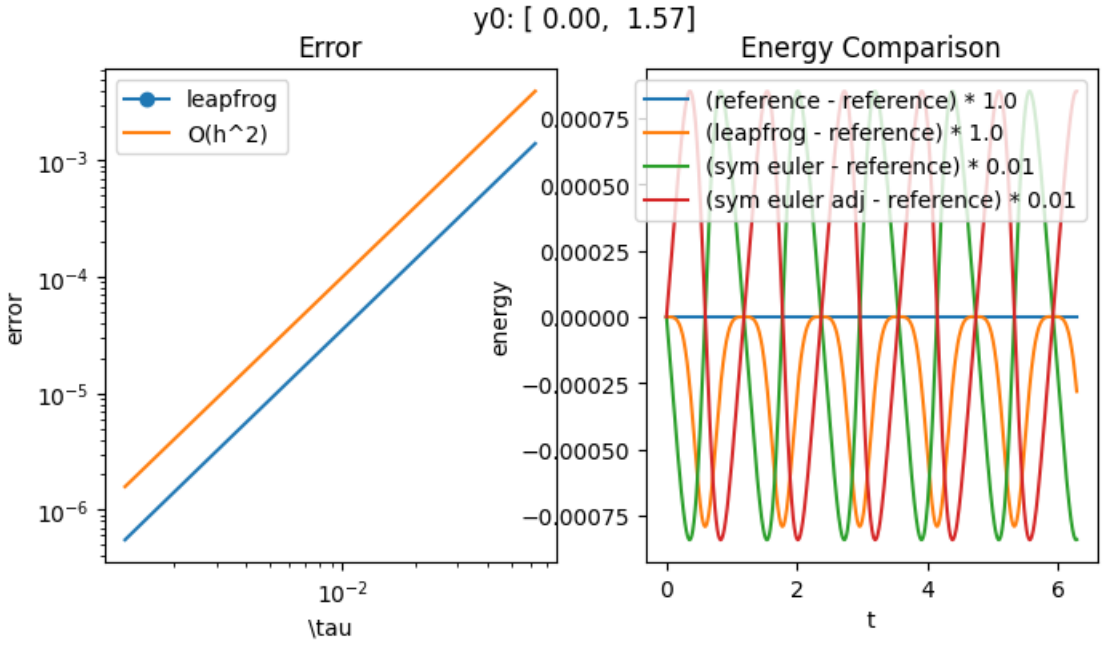
```
  Y = np.arange(y_min, y_max, 0.3).tolist()
  h = np.empty((len(Y), len(X)))
  t_eval = np.linspace(t0, T, N+1)
  for i, x in enumerate(X):
    for j, y in enumerate(Y):
      ref_e = energy(ref_sol_all(np.array([x, y]), t_eval, tspan))
      e = energy(method(np.array([x, y]), t_eval, tspan))
      h[j, i] = np.mean(e - ref_e)
  h_l.append(h)

fig, axs = plt.subplots(1, len(methods), figsize=(3 * len(methods), 4))
for i, ax in enumerate(axs):
  g = sns.heatmap(h_l[i], ax=ax)
  g.set_xticks([0, h_l[i].shape[1]-1])
  g.set_xticklabels([x_min, x_max])
  g.set_xlabel("p")
  g.set_yticks([0, h_l[i].shape[0]-1])
  g.set_yticklabels([y_min, y_max])
  g.set_ylabel("q")"""
```

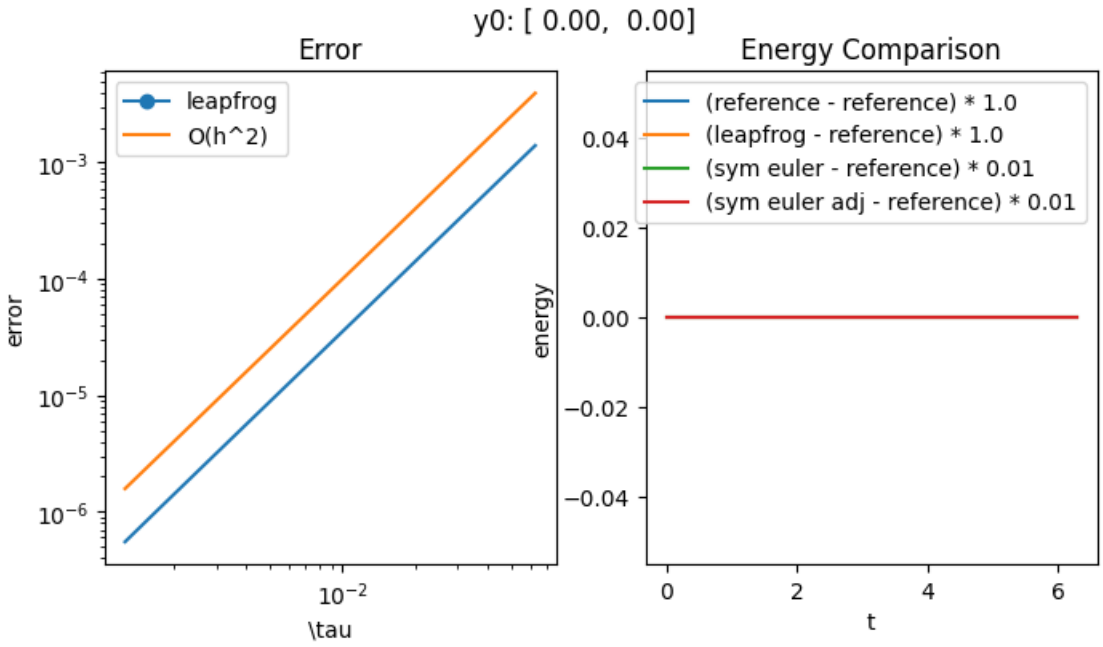
[54]:
```
'import seaborn as sns\n\nmethods = [leapfrog_all, sym_euler_all,
sym_euler_adj_all]\nh_l = []\nx_min = -3.0\nx_max = 3.0\ny_min = -np.pi\ny_max =
np.pi\nfor method in methods:\n  energy = lambda y: (1 / (2 * m * ell ** 2)) *
y[0,…] ** 2 + m * ell * g * (1 - np.cos(y[1,…]))\n  X = np.arange(x_min,
x_max, 0.3).tolist()\n  Y = np.arange(y_min, y_max, 0.3).tolist()\n  h =
np.empty((len(Y), len(X)))\n  t_eval = np.linspace(t0, T, N+1)\n  for i, x in
enumerate(X):\n    for j, y in enumerate(Y):\n      ref_e =
energy(ref_sol_all(np.array([x, y]), t_eval, tspan))\n      e =
energy(method(np.array([x, y]), t_eval, tspan))\n      h[j, i] = np.mean(e -
ref_e)\n  h_l.append(h)\n\nfig, axs = plt.subplots(1, len(methods), figsize=(3 *
len(methods), 4))\nfor i, ax in enumerate(axs):\n  g = sns.heatmap(h_l[i],
ax=ax)\n  g.set_xticks([0, h_l[i].shape[1]-1])\n  g.set_xticklabels([x_min,
x_max])\n  g.set_xlabel("p")\n  g.set_yticks([0, h_l[i].shape[0]-1])\n
g.set_yticklabels([y_min, y_max])\n  g.set_ylabel("q")'
```

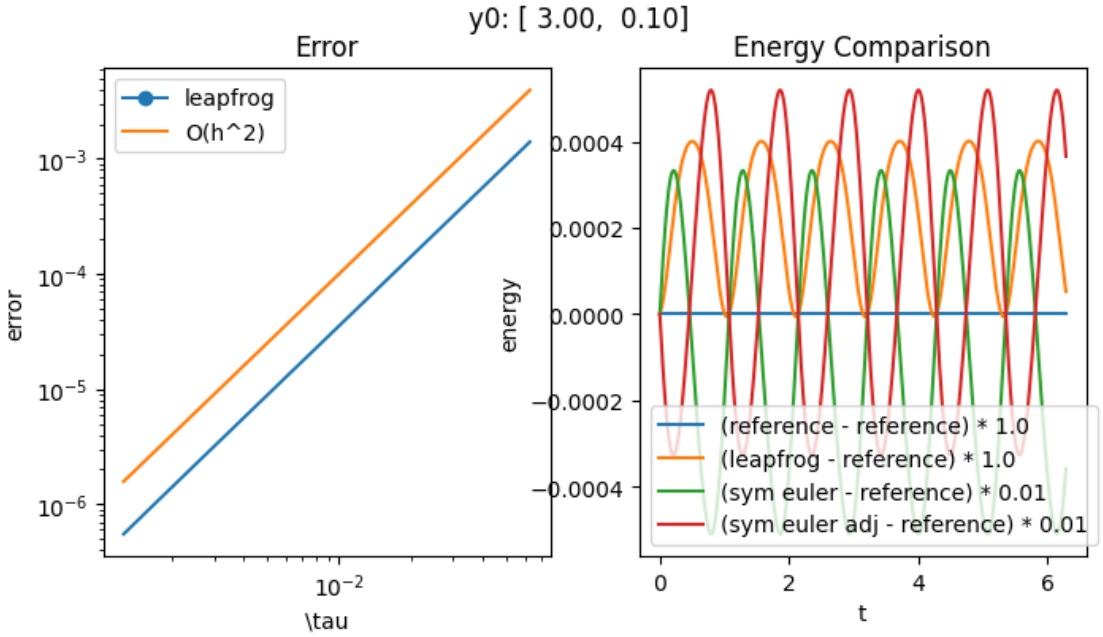

y0: [0.00, 1.00]
Error
leapfrog
O(h^2)
error
\tau
Energy Comparison
(reference - reference) * 1.0
(leapfrog - reference) * 1.0
(sym euler - reference) * 0.01
(sym euler adj - reference) * 0.01
energy
t

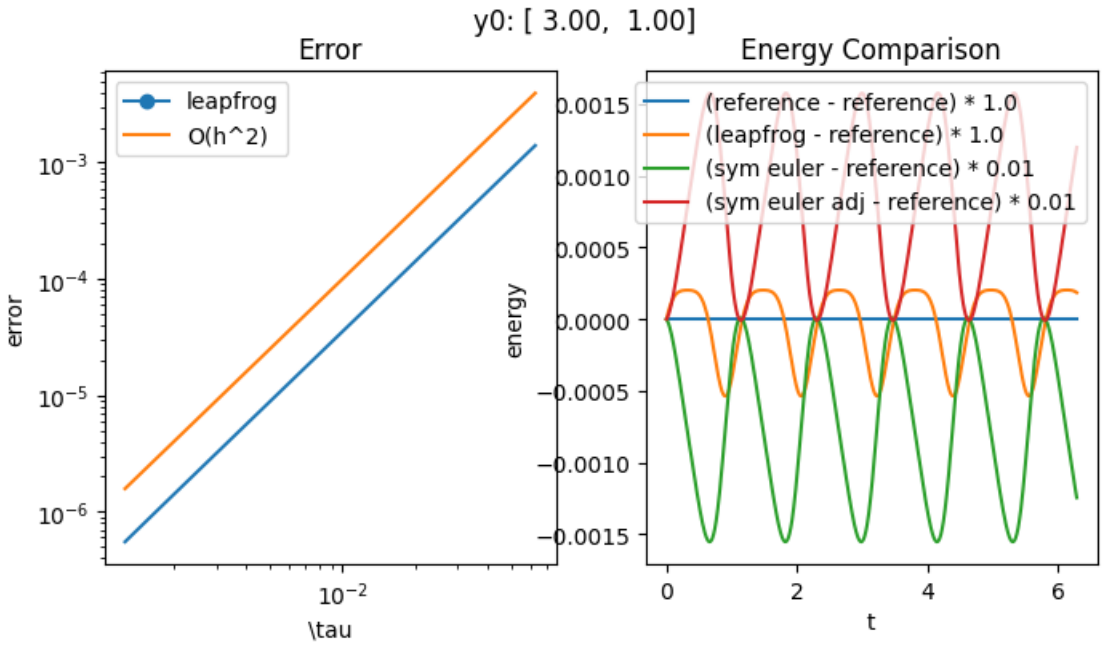

y0: [1.00, 0.00]
Error
leapfrog
O(h^2)
error
\tau
Energy Comparison
1e−5
(reference - reference) * 1.0
(leapfrog - reference) * 1.0
(sym euler - reference) * 0.01
(sym euler adj - reference) * 0.01
energy
t

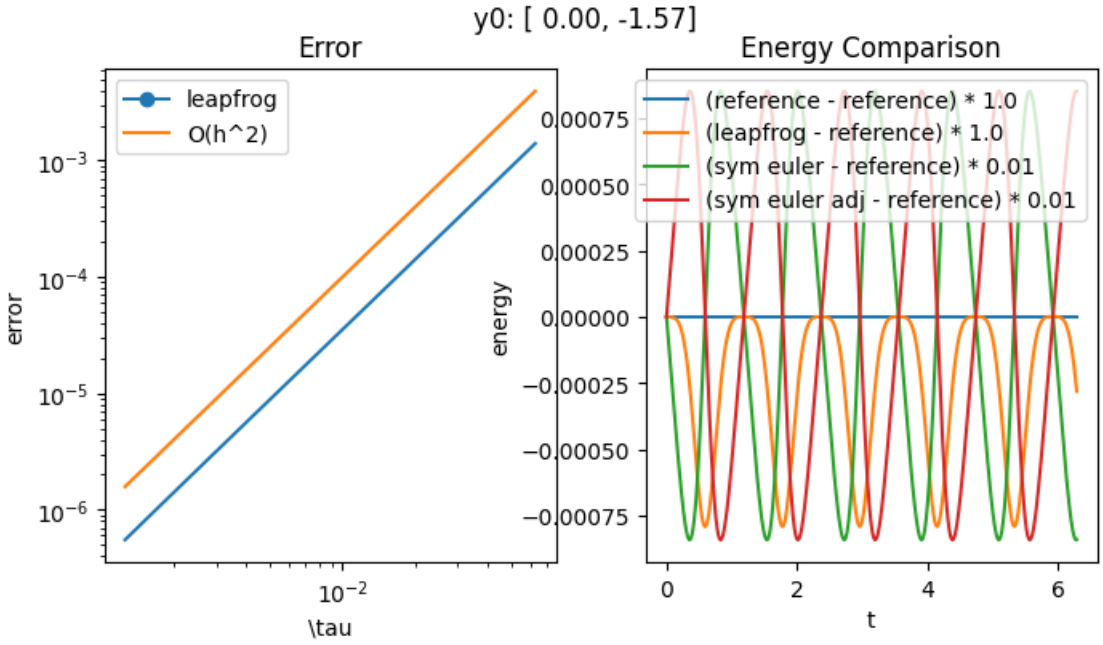

y0: [0.00, 3.14]
Error
Energy Comparison
leapfrog
O(h^2)
error
\tau
energy
t
(reference - reference) * 1.0
(leapfrog - reference) * 1.0
(sym euler - reference) * 0.01
(sym euler adj - reference) * 0.01


y0: [0.00, 1.57]
Error
Energy Comparison
leapfrog
O(h^2)
error
\tau
energy
t
(reference - reference) * 1.0
(leapfrog - reference) * 1.0
(sym euler - reference) * 0.01
(sym euler adj - reference) * 0.01


y0: [0.00, 0.00]
Error
Energy Comparison
leapfrog
O(h^2)
error
\tau
energy
t
(reference - reference) * 1.0
(leapfrog - reference) * 1.0
(sym euler - reference) * 0.01
(sym euler adj - reference) * 0.01


y0: [3.00, 0.10]
Error
Energy Comparison
leapfrog
O(h^2)
error
\tau
energy
t
(reference - reference) * 1.0
(leapfrog - reference) * 1.0
(sym euler - reference) * 0.01
(sym euler adj - reference) * 0.01





<– Platz für Ihre Antwort –>

Die Plots der Fehler stimmen mit der Ordnung 2 des Leapfrog Verfahrens überein

Die Energie bleibt gut erhalten, da das Leapfrog Verfahren als zusammengesetztes Verfahren sym-

plektischer Euler Verfahren symplektisch ist

auffällig sind die mittleren Abweichungen der Energie abhängig vom Anfangswert

3.2 b) Ein Verfahren vierter Ordnung

Ein symplektischer Integrator vierter Ordnung kann folgendermaßen konstruiert werden: Um einen Schritt der Größe τ zu machen, verwenden wir das leapfrog Verfahren dreimal hintereinander, wobei wir in Schritt 1 und 3 die Schrittweite

$$\hat{\tau}_1 = \frac{1}{2 - 2^{1/3}}\tau$$

und in Schritt 2 die Schrittweite

$$\hat{\tau}_2 = -\frac{2^{1/3}}{2 - 2^{1/3}}\tau$$

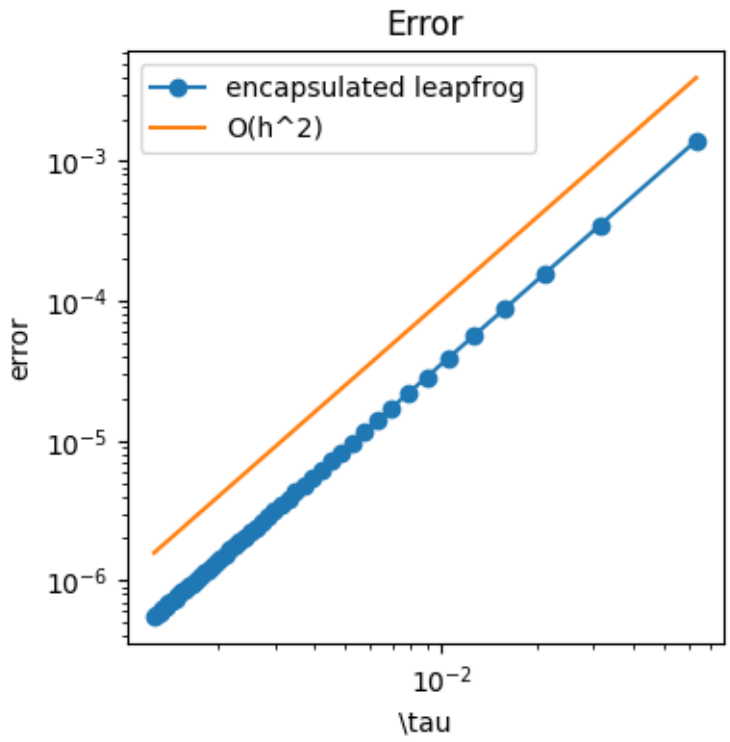
wählen. Mehr zu dieser Konstruktion finden Sie zum Beispiel in diesem paper, welches über die KIT-Bibliothek online verfügbar ist. Wir wollen nun aber nicht die neue Methode auf dem Papier aufschreiben und anschließend erneut "from scratch" implementieren, da dies sehr fehleranfällig ist. Stattdessen wollen wir zunächst das leapfrog Verfahren so umschreiben, dass es eine Funktion `LeapfrogStepPendulum()` benutzt, welche einen einzelnen Schritt mit einem gegebenen τ beschreibt. Benutzen Sie dafür folgende Vorlage

[55]:
```
def LeapfrogStep(y, tau, dTdp, dUdq):
    """
    Perform one step of the Leapfrog (Störmer-Verlet) method for a !separable!␣
 ↪Hamiltonian system.

    Parameters:
    y : current state vector [p, q]
    tau : time step size
    dTdp : function computing  T/ p
    dUdq : function computing  U/ q

    Returns:
    y_next : updated state vector after one time step
    """
    pn = y[:y.shape[0] // 2]
    qn = y[y.shape[0] // 2:]
    p_half = pn - 0.5 * tau * dUdq(qn)
    q_new = qn + tau * dTdp(p_half)
    p_new = p_half - 0.5 * tau * dUdq(q_new)
    y_next = np.concatenate((p_new, q_new), axis=0)
    return y_next



def LeapfrogIntegrate_encapsulated(y0, dTdp, dUdq, tspan, numTimeSteps=100):
    """
    Integrates the Hamiltonian system using the Leapfrog (Störmer-Verlet) method
```

```
    with encapsulated step function.

    Parameters:
    y0 : initial state [p0, q0]
    dTdp : function for  T/ p
    dUdq : function for  U/ q
    tspan : tuple (t0, T) specifying integration interval
    numTimeSteps : number of time steps

    Returns:
    times : array of time points
    states : array of states [p, q] at each time point (shape: 2 x␣
↪(numTimeSteps+1))
    """
    tau = tau_f(numTimeSteps, tspan)
    times = np.linspace(tspan[0], tspan[1], numTimeSteps+1)
    states = np.empty((y0.shape[0], numTimeSteps+1))
    states[:, 0] = y0

    for i in range(0, numTimeSteps):
      y_next = LeapfrogStep(states[:, i], tau, dTdp, dUdq)
      states[:, i+1] = y_next

    return times, states
```

Testen Sie zunächst erneut, dass das Leapfrog Verfahren funktioniert.

[56]:
```
# Initial data and parameters
q0 = 1.0
p0 = 0.0
y0 = np.array([p0, q0])
t0 = 0.0
T = 2 * np.pi
tspan = (t0, T)
N = 100
m = 1.0
ell = 1.0
g = 9.81

enapsulated_leapfrog = lambda y0, t_eval, tspan:␣
↪LeapfrogIntegrate_encapsulated(y0, dTdp, dUdq, tspan, t_eval.
↪shape[0]-1)[1][1, :]
encapsulated_leapfrog_error, NSpan = calculate_errors(ref_sol, {"encapsulated␣
↪leapfrog": enapsulated_leapfrog}, N, tspan=tspan, rhs=rhs, tol=1e-8)
plot_errors(encapsulated_leapfrog_error, NSpan, ref_f=quadratic,␣
↪ref_label="O(h^2)", combine=True, xscale="log", yscale="log")
```

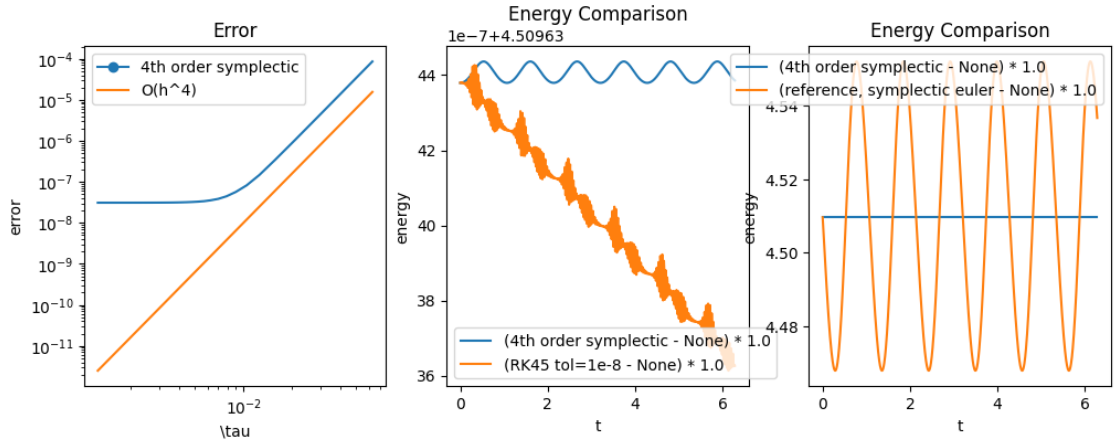


Nutzen Sie nun die Funktion `LeapfrogStepPendulum()` um mit oben erklärter Konstruktion ein Verfahren 4. Ordnung zu implementieren. Testen Sie auch dieses auf Konvergenz und Energieerhaltung. Erläutern Sie, was Ihnen im Vergleich zu den vorherigen Konvergenzplots auffällt.

[57]:
```
def FourthOrderSymplecticIntegrate(y0, dTdp, dUdq, tspan, numTimeSteps):
    """
    Fourth-order symplectic integrator (Yoshida scheme) for separable␣
 ↪Hamiltonian systems.

    Uses three nested Leapfrog (Strömer-Verlet) steps with carefully chosen␣
 ↪time fractions.

    Parameters:
    y0 : ndarray of shape (2,) - initial condition [p0, q0]
    dTdp : function - derivative of kinetic energy w.r.t. momentum
    dUdq : function - derivative of potential energy w.r.t. position
    tspan : tuple (t0, T) - time interval
    numTimeSteps : int - number of time steps

    Returns:
    times : ndarray of shape (numTimeSteps+1,) - time grid
    states : ndarray of shape (2, numTimeSteps+1) - time evolution of [p, q]
    """
```

```
    tau = tau_f(numTimeSteps, tspan)
    tau_l = [(1 / (2 - np.power(2, 1/3))) * tau,
             -np.power(2, 1/3) / (2 - np.power(2, 1/3)) * tau,
              (1 / (2 - np.power(2, 1/3))) * tau]
    times = np.linspace(tspan[0], tspan[1], numTimeSteps+1)
    states = np.empty((y0.shape[0], numTimeSteps+1))
    states[:, 0] = y0

    for i in range(0, numTimeSteps):
      yn = states[:, i]
      for tau_sub in tau_l:
        yn = LeapfrogStep(yn, tau_sub, dTdp, dUdq)
      states[:, i+1] = yn

    return times, states
```

[58]:
```
# --- Setup parameters ---
q0 = 1.0
p0 = 0.0
y0 = np.array([p0,q0])
t0 = 0.0
T = 2 * np.pi
m = 1.0
ell = 1.0
g = 9.81

# fourth reference
fourth = lambda x: np.pow(x, 4)

fourth_sym = lambda y0, t_eval, tspan: FourthOrderSymplecticIntegrate(y0, dTdp,␣
 ↪dUdq, tspan, t_eval.shape[0]-1)[1][1, :]
fourth_sym_error, numTimeStepSpan = calculate_errors(ref_sol, {"4th order␣
 ↪symplectic": fourth_sym}, 100, tspan=tspan, rhs=rhs, tol=1e-8)
er_ax = plot_errors(fourth_sym_error, numTimeStepSpan, ref_f=fourth,␣
 ↪ref_label="O(h^4)", combine=True, axes=True, xscale="log", yscale="log")
fourth_sym_all = lambda y0, t_eval, tspan: FourthOrderSymplecticIntegrate(y0,␣
 ↪dTdp, dUdq, tspan, t_eval.shape[0]-1)[1]
rk_ax = plot_energy(energy, {"4th order symplectic": fourth_sym_all, "RK45␣
 ↪tol=1e-8": ref_sol_all}, np.linspace(t0, T, 1000), y0, "Energy Comparison",␣
 ↪tspan, combine=True, axes=True)
ref_ax = plot_energy(energy, {"4th order symplectic": fourth_sym_all,␣
 ↪"reference, symplectic euler": sym_euler_all}, np.linspace(t0, T, 1000), y0,␣
 ↪"Energy Comparison", tspan, combine=True, axes=True)
plot_axs(er_ax + rk_ax + ref_ax, "", 1, 3)
```



<– Platz für Ihre Antwort –>

Man erkennt die Ordnung 4 an dem Error Plot

der konstante Fehler für hinreichend kleine Schrittweiten kann durch die Toleranz der `RK45` Referenzlösung mit Toleranz 10^{-8} erklärt werden, da der konstante Fehler fast bei 10^{-8} und dieser Fehler im Vergleich zu der Referenzlösung wenig aussagekräftig für den Fehler im Vergleich zur exakten Lösung

Die Energie wird besser erhalten als der symplektische Euler, was wegen der höheren Ordnung Sinn macht, da $ H(y_n) - H(y_0) O(^{Ordnung}) $ für exponentiell bezüglich $ 1 / $ lange Zeiten für symplektische Verfahren und das Verfahren der Ordnung 4 als zusammengesetztes Verfahren von symplektischen Euler Verfahren symplektisch

4 Aufgabe 3 - Doppelpendel

In der letzten Aufgabe dieses Notebooks wollen wir die eingeführten Verfahren auch auf das komplexere Beispiel des Doppelpendels (vgl. Beispiel 13.6) anwenden.

Wir können den Zustand des Doppelpendels durch die Vektoren

$$q = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix}, \quad \text{und} \quad p = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix}$$

beschreiben. Dabei bezeichnen q_1 und q_2 die Auslenkungswinkel der Stäbe und p_1,p_2 die konjugierten Impulse mit

$$p = \frac{\partial L}{\partial \dot{q}}(q, \dot{q}) = M(q)\dot{q}$$

mit

$$M(q) = \begin{pmatrix} (m_1 + m_2)\ell_1^2 & m_2\ell_1\ell_2\cos(q_1 - q_2) \\ m_2\ell_1\ell_2\cos(q_1 - q_2) & m_2\ell_2^2 \end{pmatrix}$$

.

Die zugehörige Hamilton-Funktion (siehe (13.20)) ist durch

$$H(p,q) = \frac{1}{2} p^T M(q)^{-1} p + U(q)$$

und

$$U(q) = -(m_1 + m_2) g l_1 \cos q_1 - m_2 g l_2 \cos q_2$$

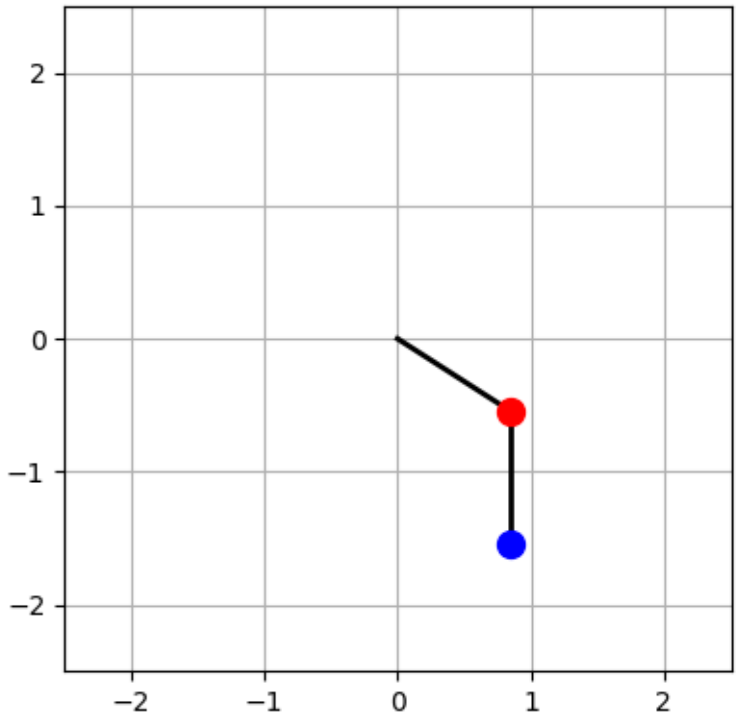
gegeben.

[59]:
```
# Parameters
g_ = 9.81                    # grav. acc.
m_1 = 1.0                    # mass 1
m_2 = 1.0                    # mass 2
l_1 = 1                      # length 1
l_2 = 1                      # length 1
# Initial state
t0 = 0                       # initial time
q1_0 = 1.0                  # initial angle (in radians)
p1_0 = 0.0                  # initial impulse
dq1_0 = 0.0                 # initial velocity
q2_0 = 0.0                  # initial angle (in radians)
p2_0 = 0.0                  # initial impulse
dq2_0 = 0.0                 # initial velocity
q0 = np.array([q1_0, q2_0])  # initial state (angles)
p0 = np.array([p1_0, p2_0])  # initial state (impulses)
dq0 = np.array([dq1_0, dq2_0])  # initial state (velocities)
```

Kennen wir q, können wir wieder die Koordinaten der Massen berechnen und das Doppelpendel mit `matplotlib` plotten. Ergänzen Sie im Folgenden die Berechnung der kartesischen Koordinaten.

[60]:
```
x1 = l_1 * np.sin(q0[0])
y1 = - l_1 * np.cos(q0[0])

x2 = x1 + l_2 * np.sin(q0[1])
y2 = y1 - l_2 * np.cos(q0[1])


# === Plot ===
fig = plt.figure(figsize=(5, 4))
ax = fig.add_subplot(autoscale_on=False)
ax.set_aspect('equal')
ax.grid()

plt.plot([0, x1], [0, y1], 'k-', lw=2)           # rod 1
plt.plot([x1, x2], [y1, y2], 'k-', lw=2)         # rod 2
plt.plot(x1, y1, 'ro', ms=10 * m_1)               # mass 1
plt.plot(x2, y2, 'bo', ms=10 * m_2)               # mass 2
plt.xlim(-l_1 - l_2 - 0.5, l_1 + l_2 + 0.5)
plt.ylim(-l_1 - l_2 - 0.5, l_1 + l_2 + 0.5)
```

```
plt.grid(True)
plt.tight_layout()
```



Wir wollen das symplektische Euler-Verfahren auf das Doppelpendel-Problem anwenden. Dazu benötigen wir die Ableitungen der Hamilton-Funktion

$$\nabla_q H(p,q), \quad \text{und} \quad \nabla_p H(p,q).$$

Da die Herleitung von Hand mühsam und fehleranfällig ist, verwenden wir stattdessen das Paket `sympy`, welches symbolische Rechnungen für uns durchführen kann. Für eine kleine Einführung zu `sympy` verweisen wir auf das Sympy-Tutorial.

Geben Sie in der folgenden Zelle die Matrix $M(q)$ und die Hamilton-Funktion $H(p,q)$ des Doppelpendels an. Anschließend können die benötigten Ableitungen von H über die Methode `sp.diff(...)` berechnet werden.

Damit wir die so symbolisch bestimmten Ableitungen später nutzen können, wandeln wir diese dann mit `sp.lambdify(...)` in von `numpy` nutzbare lambda-Funktionen um. An diesen `sp.lambdify(...)`-Aufrufen müssen Sie nichts mehr ändern.

[61]:
```
# Define symbolic variables
q1, q2, p1, p2 = sp.symbols('q1 q2 p1 p2')
q = sp.Matrix([q1, q2])
p = sp.Matrix([p1, p2])
m1, m2, l1, l2, g = sp.symbols('m1 m2 l1 l2 g')
# Define Matrix M(q)
```

```
M = sp.Matrix([[(m1 + m2) * l1 ** 2, m2 * l1 * l2 * sp.cos(q1 - q2)],
               [m2 * l1 * l2 * sp.cos(q1 - q2), m2 * l2 ** 2]])
# Define Hamiltonian components
U = - (m1 + m2) * g * l1 * sp.cos(q1) - m2 * g * l2 * sp.cos(q2)
T = 0.5 * p.T * (M ** -1) * p
H = T[0, 0] + U
# Derivative of H with respect to p (this gives q_dot)
dH_dp = sp.diff(H, p)
# Derivative of H with respect to q (this gives -p_dot)
dH_dq = sp.diff(H, q)

# Lambdify to get functions for numerical evaluation
Minv_func = sp.lambdify((*q, m1, m2, l1, l2), M.inv(), "numpy")
H_func = sp.lambdify((*p, *q, m1, m2, l1, l2, g), H, "numpy")
dH_dp_func = sp.lambdify((*p, *q, m1, m2, l1, l2, g), dH_dp, "numpy")
dH_dq_func = sp.lambdify((*p, *q, m1, m2, l1, l2, g), dH_dq, "numpy")
```

Mithilfe der Funktion `functools.partial` können wir die so erzeugten lambdas auch teilweise vorauswerten.

[62]:
```
from functools import partial
Minvfunc_dPendulum = partial(Minv_func, m1=m_1, m2=m_2, l1=l_1, l2=l_2)
Hfunc_dPendulum = partial(H_func, m1=m_1, m2=m_2, l1=l_1, l2=l_2, g=g_)
# do the same for dHdp_dPendulum and dHdq_dPendulum
dHdp_dPendulum = partial(dH_dp_func, m1=m_1, m2=m_2, l1=l_1, l2=l_2, g=g_)
dHdq_dPendulum = partial(dH_dq_func, m1=m_1, m2=m_2, l1=l_1, l2=l_2, g=g_)
# small test -> should return array([[16.50966072],[0.]])
dHdq_dPendulum(*[p1_0,p2_0,q1_0,q2_0])
```

[62]:
```
array([[16.50966072],
       [ 0.        ]])
```

Wenden Sie nun das symplektische Euler-Verfahren auf das Doppelpendel an. Nutzen Sie dazu die Funktions-lambdas, welche Sie mit `sympy` erzeugt haben. Orientieren Sie sich an der Funktion `BackwardEulerIntegratePendulum(...)` aus Aufgabe 1. Sie können wieder `scipy.optimize.fsolve(...)` zur Lösung des nichtlinearen Gleichungssystems verwenden.

[63]:
```
def SymplecticEulerIntegrate_NonSep(y0, dHdp, dHdq, tspan, numTimeSteps=100):
    """
    Symplectic Euler integrator for a non-separable Hamiltonian system.

    Parameters:
    y0 : initial state [p1, p2, q1, q2]
    dHdp : function returning  H/ p as array([dq1/dt, dq2/dt]) given full state
    dHdq : function returning  H/ q as array([dp1/dt, dp2/dt]) given full state
    tspan : tuple (start_time, end_time)
    numTimeSteps : number of time steps
```

```
    Returns:
    times : array of time points
    states : array of shape (4, numTimeSteps+1), each column is [p1, p2, q1,␣
 ↪q2] at a time step
    """

    times = np.linspace(tspan[0], tspan[1], numTimeSteps+1)
    states = np.empty((4, numTimeSteps+1))
    states[:, 0] = y0

    tau = tau_f(numTimeSteps, tspan)

    def solve_pnew_implicit(new, n, q):
      return n - new - tau * np.squeeze(dHdq(new[0], new[1], q[0], q[1]))

    for i in range(0, numTimeSteps):
      pn = states[0:2, i]
      qn = states[2:, i]
      estimate = pn
      pnew = fsolve(solve_pnew_implicit, estimate, args=(pn, qn,))
      qnew = qn + tau * np.squeeze(dHdp(pnew[0], pnew[1], qn[0], qn[1]))
      states[0:2, i+1] = pnew
      states[2:, i+1] = qnew


    return times, states
```

Simulieren Sie das Pendel auf dem Zeitintervall $[0, 10]$ mit 1000 Zeitschritten. Visualiseren Sie anschließend das Doppelpendel in einem interaktiven Plot.

[64]:
```
times_for_ref, states_sym = SymplecticEulerIntegrate_NonSep(np.concatenate((p0,␣
 ↪q0)), dHdp_dPendulum, dHdq_dPendulum, (0, 10), 1000)
```

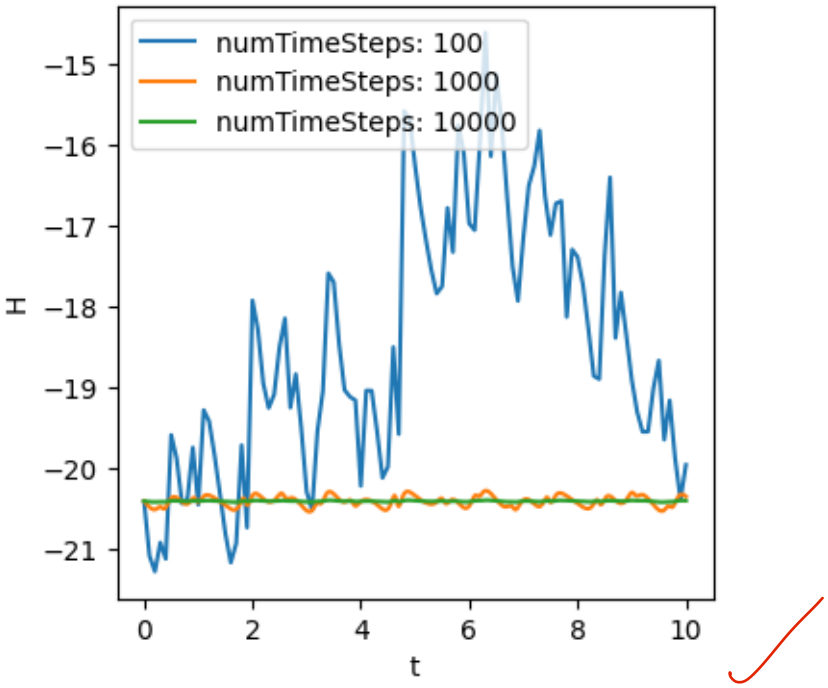
[65]:
```
@interact(n=widgets.IntSlider(min=0, max=len(times_for_ref)-1, step=1, value=0))
def visualize_double_pendulum(n):
  q_sym = states_sym[2:, n]
  x1 = l_1 * np.sin(q_sym[0])
  y1 = - l_1 * np.cos(q_sym[0])

  x2 = x1 + l_2 * np.sin(q_sym[1])
  y2 = y1 - l_2 * np.cos(q_sym[1])

  fig = plt.figure(figsize=(5, 4))
  ax = fig.add_subplot(autoscale_on=False)
  ax.set_aspect('equal')
  ax.grid()
```

```
  plt.plot([0, x1], [0, y1], 'k-', lw=2)           # rod 1
  plt.plot([x1, x2], [y1, y2], 'k-', lw=2)         # rod 2
  plt.plot(x1, y1, 'ro', ms=10 * m_1)               # mass 1
  plt.plot(x2, y2, 'bo', ms=10 * m_2)               # mass 2
  plt.xlim(-l_1 - l_2 - 0.5, l_1 + l_2 + 0.5)
  plt.ylim(-l_1 - l_2 - 0.5, l_1 + l_2 + 0.5)
  plt.grid(True)
  plt.tight_layout()
```

```
interactive(children=(IntSlider(value=0, description='n', max=1000), Output()),␣
 ↪_dom_classes=('widget-interact…
```

Werten Sie die Hamilton-Funktion H an jedem Zeitschritt aus. Wie sollte sich das exakte H über die Zeit verhalten? Wie ändert sich das Schaubild, wenn Sie mehr oder weniger Zeitschritte verwenden?

[66]:
```
numTimeSteps_l = [100, 1000, 10000]
fig = plt.subplots(figsize=(4, 4))
for numTimeSteps in numTimeSteps_l:
  times, states = SymplecticEulerIntegrate_NonSep(np.concatenate((p0, q0)),␣
 ↪dHdp_dPendulum, dHdq_dPendulum, (0, 10), numTimeSteps)
  plt.plot(times, Hfunc_dPendulum(states[0, :], states[1, :], states[2, :],␣
 ↪states[3, :]), marker="", linestyle="-", label=f"numTimeSteps:␣
 ↪{numTimeSteps}")
plt.xlabel("t")
plt.ylabel("H")
plt.legend()
plt.show()
```



<– Platz für Ihre Antwort –>

Das exakte $ H $ soll konstant bleiben, da es ein erstes Integral der DGL ist

Die Amplitude der $ H $ - Werte des symplektischen Euler Verfahrens nimmt mit zunehmender Anzahl an Schritten ab, was dazu passt, dass $ H(y_n) - H(y_0) O( ) $ für exponentiell bezüglich $ 1 / $ große Zeiten

Berechnen Sie über die Gleichung (13.17b)

$$\begin{pmatrix} \dot{q} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ M^{-1}(q) f(q, v) \end{pmatrix}$$

eine Referenzlösung mithilfe des `RK45` Verfahrens. Bestimmen Sie dazu zunächst $M(q)^{-1}$ und $f(q, v)$ und erstellen Sie eine neue rechte Seite `rhs_DoublePendulum`. Nutzen Sie dann die Funktion `referenceSolutionRK45`, welche Sie bereits oben angelegt haben. Legen Sie dann eine rechte Seite für das Doppelpendel an und nutzen Sie `solve_ivp`, um die Lösung der Differentialgleichung zu approximieren. Wählen Sie als Endzeit $T = 10$ und werten Sie die Approximation an den Zeitschritten des symplektischen Eulers aus.

[67]:
```
dU_dq = sp.diff(U, q)
dU_dq_func = sp.lambdify((*q, m1, m2, l1, l2, g), dU_dq, "numpy")
dUdq_dPendulum = partial(dU_dq_func, m1=m_1, m2=m_2, l1=l_1, l2=l_2, g=g_)

def f(q,v):
  alpha = m_2 * l_2 * l_1 * np.sin(q[0] - q[1])
```

```
  return alpha * np.array([[0, -1], [1, 0]]) @ (v ** 2) - np.
 ↪squeeze(dUdq_dPendulum(q[0], q[1]))

def rhs_DoublePendulum(t,y):
    q = y[0:2]
    v = y[2:]
    return np.concatenate((v, Minvfunc_dPendulum(q[0], q[1]) @ f(q, v)))

# Be careful, what is now y0?
#def referenceSolutionRK45(y0, rhs, tspan, times, tol=1e-8):
#    ...
```

Visualiseren Sie auch die Referenzlösung.

[68]:
```
v = lambda q, p: Minvfunc_dPendulum(q[0], q[1]) @ p
v0 = v(q0, p0)
ref_sol = referenceSolutionRK45(np.concatenate((q0, v0)), rhs_DoublePendulum,␣
 ↪(0, 10), times_for_ref, tol=1e-8)

@interact(n=widgets.IntSlider(min=0, max=len(times_for_ref)-1, step=1, value=0))
def visualize_double_pendulum(n):
  q_ref = ref_sol[0:2, n]
  x1 = l_1 * np.sin(q_ref[0])
  y1 = - l_1 * np.cos(q_ref[0])

  x2 = x1 + l_2 * np.sin(q_ref[1])
  y2 = y1 - l_2 * np.cos(q_ref[1])

  fig = plt.figure(figsize=(5, 4))
  ax = fig.add_subplot(autoscale_on=False)
  ax.set_aspect('equal')
  ax.grid()

  plt.plot([0, x1], [0, y1], 'k-', lw=2)           # rod 1
  plt.plot([x1, x2], [y1, y2], 'k-', lw=2)         # rod 2
  plt.plot(x1, y1, 'ro', ms=10 * m_1)               # mass 1
  plt.plot(x2, y2, 'bo', ms=10 * m_2)               # mass 2
  plt.xlim(-l_1 - l_2 - 0.5, l_1 + l_2 + 0.5)
  plt.ylim(-l_1 - l_2 - 0.5, l_1 + l_2 + 0.5)
  plt.grid(True)
  plt.tight_layout()
```

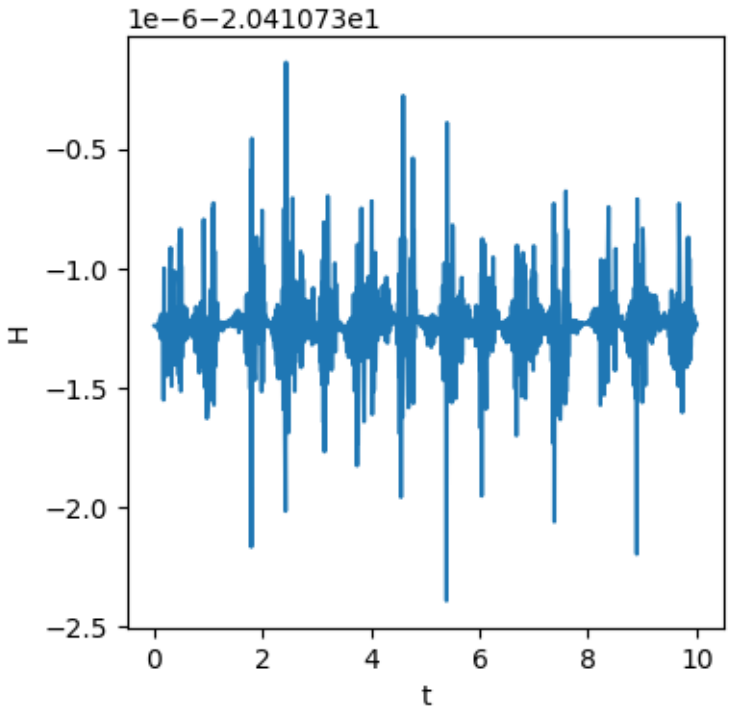
```
interactive(children=(IntSlider(value=0, description='n', max=1000), Output()),␣
 ↪_dom_classes=('widget-interact…
```

Berechnen Sie die zugehörigen konjugierten Impulse

$$p = \frac{\partial L}{\partial \dot{q}}(q, \dot{q}) = M(q)\dot{q}$$

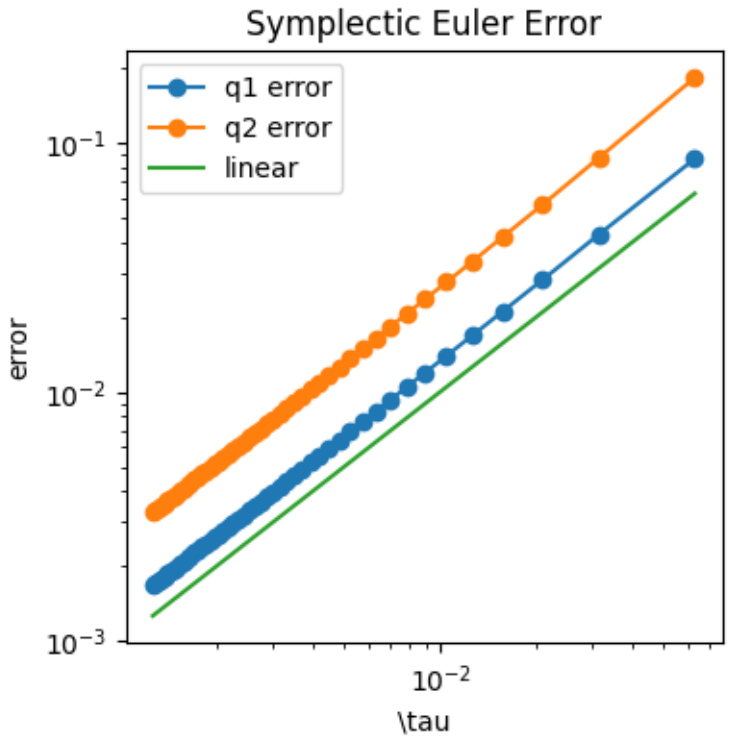
und werten Sie auch unter Verwendung der Referenzlösung die Hamilton-Funktion H über die Zeit aus.

[69]:
```
M_func = sp.lambdify((*q, m1, m2, l1, l2, g), M, "numpy")
Mfunc_dPendulum = partial(M_func, m1=m_1, m2=m_2, l1=l_1, l2=l_2, g=g_)

conj_imp = lambda q, v: Mfunc_dPendulum(q[0], q[1]) @ v

p_states = np.concatenate([np.expand_dims(conj_imp(ref_sol[0:2, i], ref_sol[2:,␣
 ↪i]), 1) for i in range(0, ref_sol.shape[1])], axis=1)
fig = plt.figure(figsize=(4, 4))
plt.xlabel("t")
plt.ylabel("H")
plt.plot(times_for_ref, Hfunc_dPendulum(p_states[0, :], p_states[1, :],␣
 ↪ref_sol[0, :], ref_sol[1, :]), marker="", linestyle="-")
plt.show()
```



Erstellen Sie einen Konvergenzplot für das symplektische Eulerverfahren angewandt auf das Doppelpendel. Nutzen Sie als Fehlermaß die maximale absolute Winkelabweichung zwischen der Approximierten mit dem symplektischen Euler und der Referenzlösung über alle Zeitschritte in beiden Winkeln q_1 und q_2. Zeichnen Sie je eine Linie für den Fehler in q_1 und für den Fehler in q_2. Nutzen Sie einen logarithmisch skalierten Plot und ergänzen Sie auch eine Linie mit der Ordnung 1.

[70]:
```
ref_sol_dPendulum = lambda y0, t_eval, tspan: referenceSolutionRK45(np.
 ↪concatenate((y0[2:], v(y0[2:], y0[0:2]))), rhs_DoublePendulum, tspan,␣
 ↪t_eval)[0:2, :]
sym_euler_dPendulum = lambda y0, t_eval, tspan:␣
 ↪SymplecticEulerIntegrate_NonSep(y0, dHdp_dPendulum, dHdq_dPendulum, tspan,␣
 ↪t_eval.shape[0]-1)[1][2:, :]
q1_error = lambda q_ref, q: np.max(np.abs(q_ref[0, :] - q[0, :]))
q2_error = lambda q_ref, q: np.max(np.abs(q_ref[1, :] -  q[1, :]))
y0 = np.concatenate((p0, q0))

integrators_dPendulum = {"symplectic euler": sym_euler_dPendulum}
q1_errors, q1_numTimeStepspan = calculate_errors(ref_sol_dPendulum,␣
 ↪integrators_dPendulum, 100, tspan=tspan, error_function=q1_error, y0=y0)
q2_errors, q2_numTimeStepSpan = calculate_errors(ref_sol_dPendulum,␣
 ↪integrators_dPendulum, 100, tspan=tspan, error_function=q2_error, y0=y0)
errors = {"q1 error": list(q1_errors.values())[0], "q2 error": list(q2_errors.
 ↪values())[0]}
q1_error_axs = plot_errors(errors, q1_numTimeStepspan, combine=True,␣
 ↪title="Symplectic Euler Error", xscale="log", yscale="log")
```



[70]: