lab01

March 16, 2024

1 Labor 01 – Grundlagen

Referenzimplementierung

Dies ist ein Jupyter Notebook. Sie könnnen Jupyter Notebook starten mit Hilfe des Befehls

```
$ jupyter notebook
```

(siehe Aufgabe 1.1). Anschließend öffnet sich ein Browser, und Sie können dieses Notebook laden und interaktiv ausführen.

Alternativ können Sie Visual Studio Code nutzen um mit Jupyter Notebooks zu arbeiten, wie in diesem Artikel beschrieben.

1.1 Kurze Einführung in Python

Python ist eine beliebte und weit verbreitete Programmiersprache. Die Sprache wird nicht kompiliert, sondern interpretiert.

Die Syntax ist entfernt verwandt mit der von C.

Im Gegensatz zu C werden bei Python die Typen nicht explizt angegeben. Vielmehr nutzt Python *duck typing*.

Um Variablen zuzuweisen, nutzen Sie `=`, z.B.

```
a = 5.0
```

Sie können auch mehrere Variablen gleichzeitig zuweisen, z.B.

```
a, b = 5.0, 7.0
```

Funktionen werden mit Hilfe des Schlüsselwortes `def` definiert. Der Rückgabewert wird – wie in C – mit Hilfe des Schlüsselwortes `return` zurückgegeben. In Python kann eine Funktion mehrere Werte zurückgeben.

Beispiel: Implementierung der Eulerschen Formel.

```
def euler(theta):
    s, c = sin(theta), cos(theta)
    return s, c
```

Der Funktionsrumpf wird nicht wie in C durch geschweifte Klammern `{}` gekennzeichnet, sondern durch *Einrückung*. Dies gilt generell für jeden *Scope*, also auch z.B. für `if` statements.

Python bietet eine Vielzahl an Bibliotheken, in denen Sie häufig genutzte Funktionen und Klassen finden. Sie kommen in Form sogenannter Packages und werden mit `import` importiert (das entspricht grob dem `#include` aus C).

Bevor Sie eine bestimmte Funktionalität selber implementieren, sollten Sie (z.B. unter Nutzung von Google) nachsehen, ob es diese Funktionalität nicht bereits in einem Package gibt. Ausnahme: Sie wollen *verstehen*, wie ein Algorithmus funktioniert – in dem Fall ergibt es Sinn, ihn einmal selber zu implementieren.

Nicht zuletzt wurden die Beliebtheit und Verbreitung von Python durch die massiven Fortschritte im Bereich *Deep Learning* angefacht, die wir in der letzten Dekade (2010-2020) gesehen haben. Als Nebeneffekt entstanden sehr viele Tutorials, die Ihnen einen schnellen Start in die Grundlagen der Sprache vermitteln. Ein Startpunkt ist z.B. die offizielle offizielle Python Seite.

1.2 Erstellung von Plots

1.2.1 Scatterplots

Erstellen Sie einen Scatterplot der folgenden komplexen Zahlen aus Übung 1, einmal für $r_0 = 2$ und $\theta_0 = \pi/4$, und einmal für $r_0 = 2$ und $\theta_0 = \pi/2$.

- $z_1 = r_0 e^{-j\theta_0}$
- $z_2 = r_0$
- $z_3 = r_0 e^{j(\theta_0+\pi)}$
- $z_4 = r_0 e^{j(-\theta_0+\pi)}$
- $z_5 = r_0 e^{j(\theta_0+2\pi)}$

Zuerst importieren wir die benötigten Pakete. Die letzten beiden Zeilen sorgen dafür, dass die Graphiken im PDF-Export als Vektorgraphiken dargestellt werden – sie haben keine inhaltliche Bedeutung für dieses Labor und können ignoriert werden.

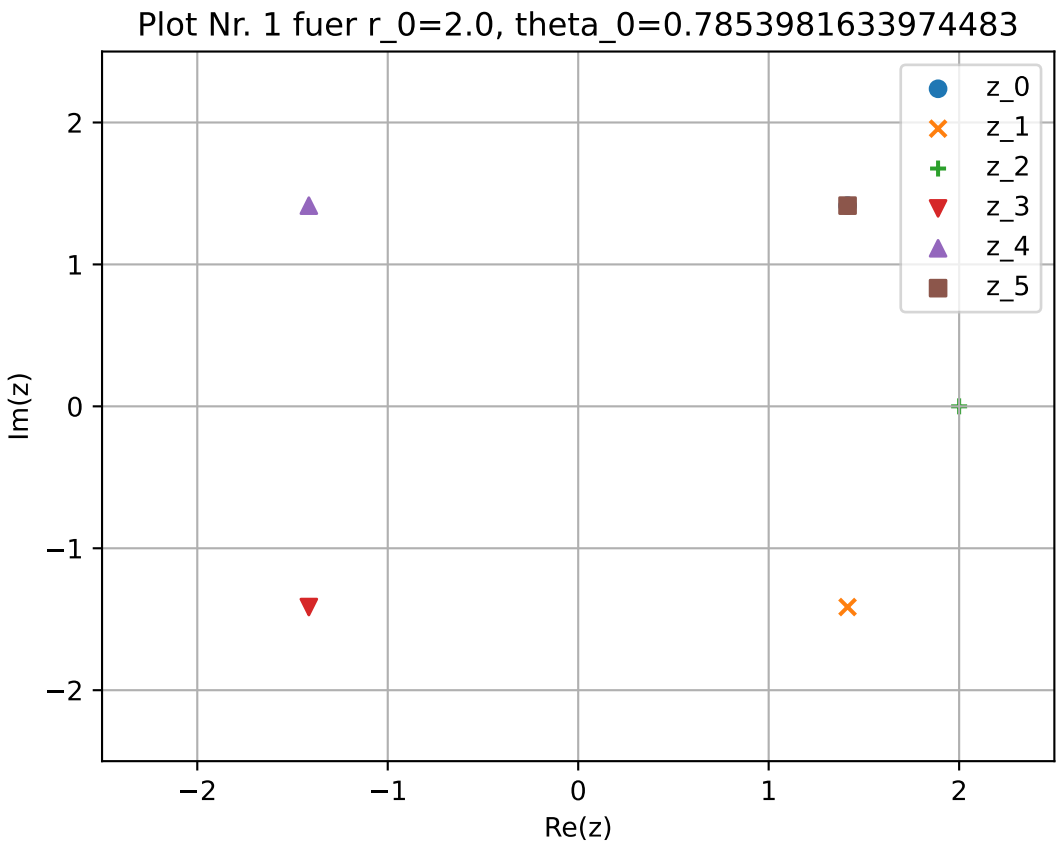
[1]:
```
import matplotlib.pyplot as plt
from numpy import real, imag, arctan2, hypot, cos, sin, pi
from numpy import linspace, array, exp

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
```

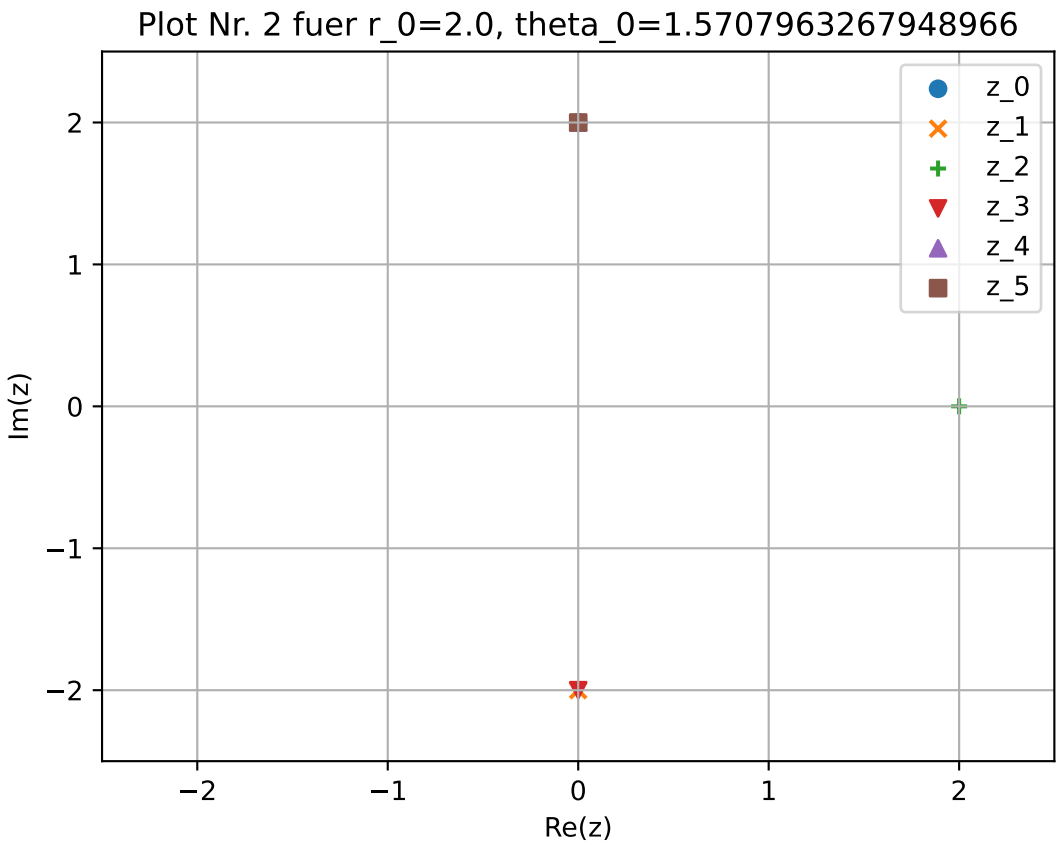
```
/var/folders/_h/ph8s95tn6nxfj5lc1njcg31h0000gn/T/ipykernel_2104/587532209.py:6:
DeprecationWarning: `set_matplotlib_formats` is deprecated since IPython 7.23,
directly use `matplotlib_inline.backend_inline.set_matplotlib_formats()`
  set_matplotlib_formats('png', 'pdf')
```

Nun können wir die Scatterplots erzeugen. Hierzu nutzen wir die Funktion `scatter`, die uns von `pyplot`, welches wir unter dem Alias `plt` importiert haben, zur Verfügung gestellt wird.

[2]:
```
r_0 = [2.0, 2.0]
theta_0 = [pi/4.0, pi/2.0]

def polar_to_cartesian(r, theta):
    # Eulerformel
```

```
    x, y = cos(theta), sin(theta)
    # Skalieren mit Betrag r
    return [r*x, r*y]

for k, (r, theta) in enumerate(zip(r_0, theta_0)):
    z0 = polar_to_cartesian(r, theta)
    z1 = polar_to_cartesian(r, -theta)
    z2 = polar_to_cartesian(r, 0)
    z3 = polar_to_cartesian(r, theta+pi)
    z4 = polar_to_cartesian(r, -theta+pi)
    z5 = polar_to_cartesian(r, theta+2*pi)

    plt.figure(k)
    plt.clf()
    plt.scatter(z0[0], z0[1], label='z_0')
    plt.scatter(z1[0], z1[1], label='z_1', marker='x')
    plt.scatter(z2[0], z2[1], label='z_2', marker='+')
    plt.scatter(z3[0], z3[1], label='z_3', marker='v')
    plt.scatter(z4[0], z4[1], label='z_4', marker='^')
    plt.scatter(z5[0], z5[1], label='z_5', marker='s')
    plt.grid(True)
    plt.legend()
    plt.xlabel('Re(z)')
    plt.ylabel('Im(z)')
    plt.xlim(-2.5,2.5)
    plt.ylim(-2.5,2.5)
    plt.title(f'Plot Nr. {k+1} fuer r_0={r}, theta_0={theta}')
```


Plot Nr. 1 fuer r_0=2.0, theta_0=0.7853981633974483
z_0
z_1
z_2
z_3
z_4
z_5
Im(z)
Re(z)
2
1
0
−1
−2
−2
−1
0
1
2



1.2.2 Kurven

Stellen Sie den Verlauf der Funktion $y(t)$ dar, welche die folgenden Differentialgleichungen löst.

- $\dot{y}_1(t) + y_1(t) = 0$ mit der Anfangsbedingung $y_1(0) = 1$
- $\ddot{y}_2(t) + \omega^2 y_2(t) = 0$ mit den Anfangsbedingungen $y_2(0) = 0, \dot{y}_2(0) = \omega$ und $\omega = 2\pi$.

Sie haben die Lösung bereits in Übung 1, Teilaufgabe 1.4.3, zu

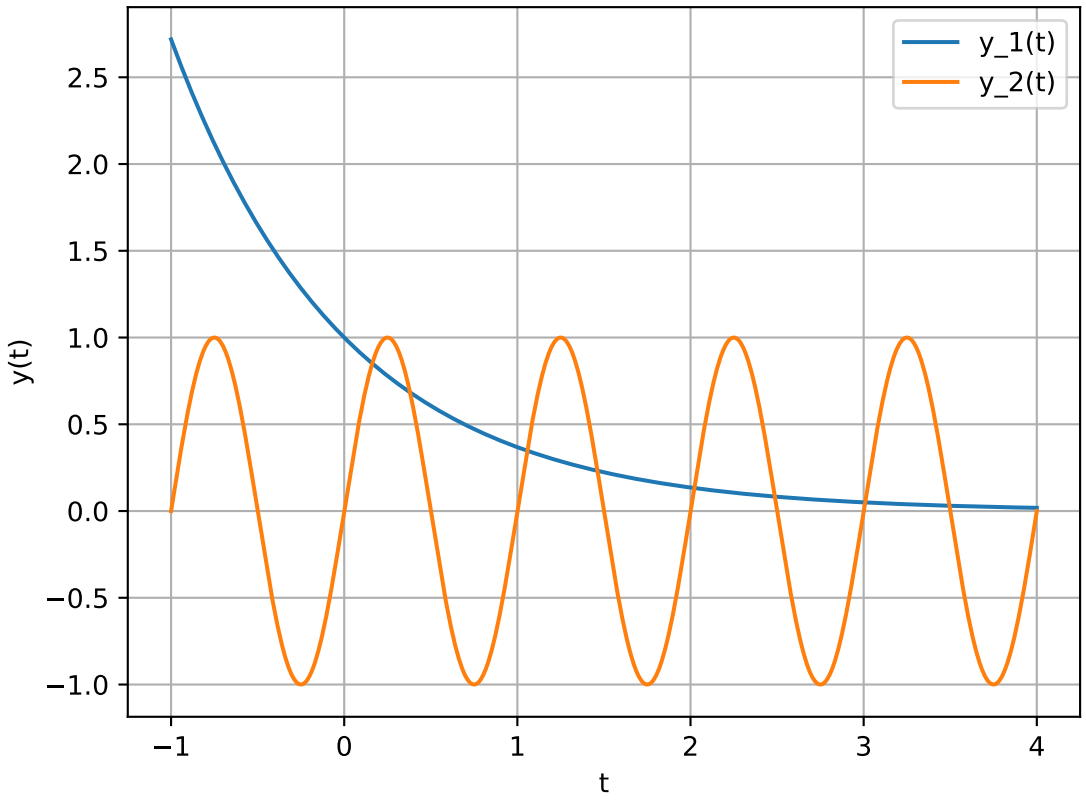
- $y_1(t) = e^{-t}$
- $y_2(t) = \sin(\omega t)$

berechnet.

Wir nutzen hierfür die Funktion `plot`. Die Achsenbeschriftungen lassen sich mit `xlabel` und `ylabel` hinzufügen.

[3]:
```
t = linspace(-1,4,1000)
y_1 = exp(-t)
omega = 2*pi
y_2 = sin(omega*t)

plt.figure(10)
```

```
plt.clf()
plt.plot(t, y_1, label='y_1(t)')
plt.plot(t, y_2, label='y_2(t)')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.grid(True)
plt.legend()
plt.show()
```



1.3 Numerische Lösung von Differentialgleichungen

1.3.1 Implementierung des "Euler vorwärts" Verfahrens

In Übung 1, Teilaufgabe 1.4.3, haben Sie das "Euler vorwärts" Verfahren kennen gelernt. Die Idee bestand darin, den Differentialquotienten durch den Differenzenquotienten zu approximieren. Dadurch konnte eine approximative Lösung der Differentialgleichung numerisch berechnet werden.

1.3.2 Nutzung von SciPy

Das `scipy` Package bietet ihnen ausgereifte Funktionen zur numerischen Lösung von Differentialgleichungen. Hierfür können Sie `scipy.integrate.solve_ivp` nutzen. IVP bedeutet *initial value problem*, d.h. dass wir die Lösung einer Differentialgleichung suchen, deren *Anfangsbedingungen*

wir kennen. Sie werden den Begriff ODE lesen: Dieser steht für *ordinary differential equation*. Die im Rahmen dieser Veranstaltung behandelten Differentialgleichungen sind ODEs. Partielle Differentialgleichungen, also solche in denen partielle Ableitungen auftreten, kommen im Rahmen dieses Kurses nicht vor.

Verwenden Sie die Funktion `solve_ivp` zur numerischen Lösung der beiden oben genannten Differentialgleichungen. Vergleichen Sie die Lösung mit Ihrer "Euler vorwärts" Lösung sowie mit der analytischen Lösung.

Wir importieren zuerst wieder die notwendigen Pakete.

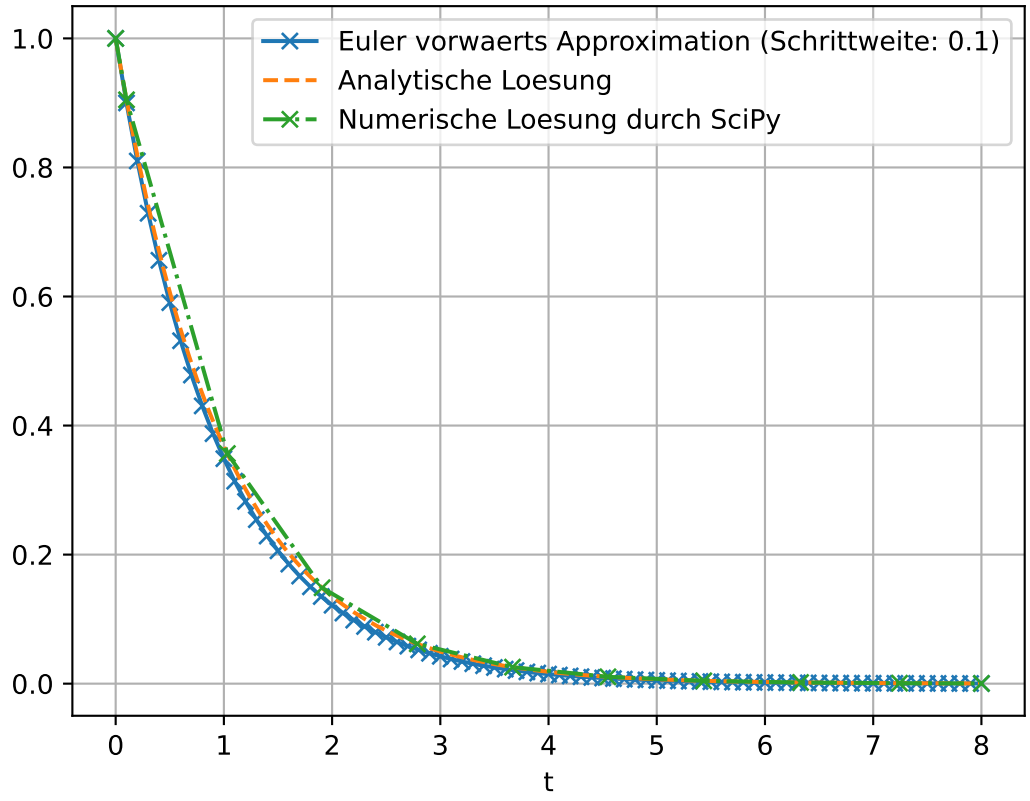
[4]:
```
import matplotlib.pyplot as plt
from numpy import exp, linspace
from numpy import array, matmul, pi, eye, sin
from scipy.integrate import solve_ivp
```

[5]:
```
# Euler vorwaerts fuer die erste Differentialgleichung

# Rechte Seite der Differenzengleichung
def next_x(x_k, step_size):
    x_k_plus_1 = (1.0 - step_size) * x_k
    return x_k_plus_1

# Anfangsbedingung
x_0 = 1.0

# Schrittweite festlegen
delta_t = 0.1

# Ende der Integration (in s)
t_f = 8.0

x = []
t = []

# Numerische Integration durchfuehren
current_t = 0.0
current_x = x_0
while current_t <= t_f:
    t.append(current_t)
    x.append(current_x)
    current_x = next_x(current_x, delta_t)
    current_t += delta_t
```

[6]:
```
# Analytische Loesung
t_analytic = linspace(0.0, t_f, 1000)
x_analytic = [exp(-tau) for tau in t_analytic]
```

[7]:
```
# Numerische Integration mit SciPy

# Definition der rechten Seite (right hand side, rhs) der
# Differentialgleichung (ordinary differential equation, ODE)
# dx/dt = f(t, x)
def ode_rhs(t, x):
    x_dot = -x
    return x_dot

scipy_integration_result = solve_ivp(ode_rhs, [0, t_f], array([x_0]))
```

[8]:
```
# Ergebnisse darstellen
plt.figure(1)
plt.clf()
plt.plot(t, x, '-x', label=f'Euler vorwaerts Approximation (Schrittweite:␣
 ↪{delta_t})')
plt.plot(t_analytic, x_analytic, '--', label='Analytische Loesung')
plt.plot(scipy_integration_result.t, scipy_integration_result.y.flatten(), '-.
 ↪x', label='Numerische Loesung durch SciPy')
plt.grid(True)
plt.legend()
plt.xlabel('t')
plt.show()
```



Wie wählen Sie die Schrittweite Δt? Begründen Sie Ihre Antwort!

Was passiert, wenn Sie die Schrittweite sehr groß wählen? Haben Sie eine Erklärung für Ihre Beobachtung?

Antwort: Die Schrittweite muss klein genug gewählt werden, da sonst die Integration instabil wird. Konkret passiert das für delta_t > 2.0.

Anschauliche Erklärung: Wir schießen zu weit auf die andere Seite, dadurch schwingt sich das System auf (die Lösung, die wir durch die numerische Integration erhalten ist instabil, obwohl das System eigentlich stabil ist!).

Mathematische Erklärung via z-Transformation.

Wenn die Schrittweite zu klein gewählt wird erhöht sich der Rechenaufwand unnötig.

Was fällt Ihnen auf?

Antwort: Obwohl SciPy eine deutlich größere Schrittweite wählt (erkennbar an den Kreuzen), ist die Lösung deutlich näher an der analytischen Lösung. Bei der zweiten Differentialgleichung (siehe unten) wird dadurch das Aufschwingen vermieden, das wir mit Euler vorwärts beobachten.

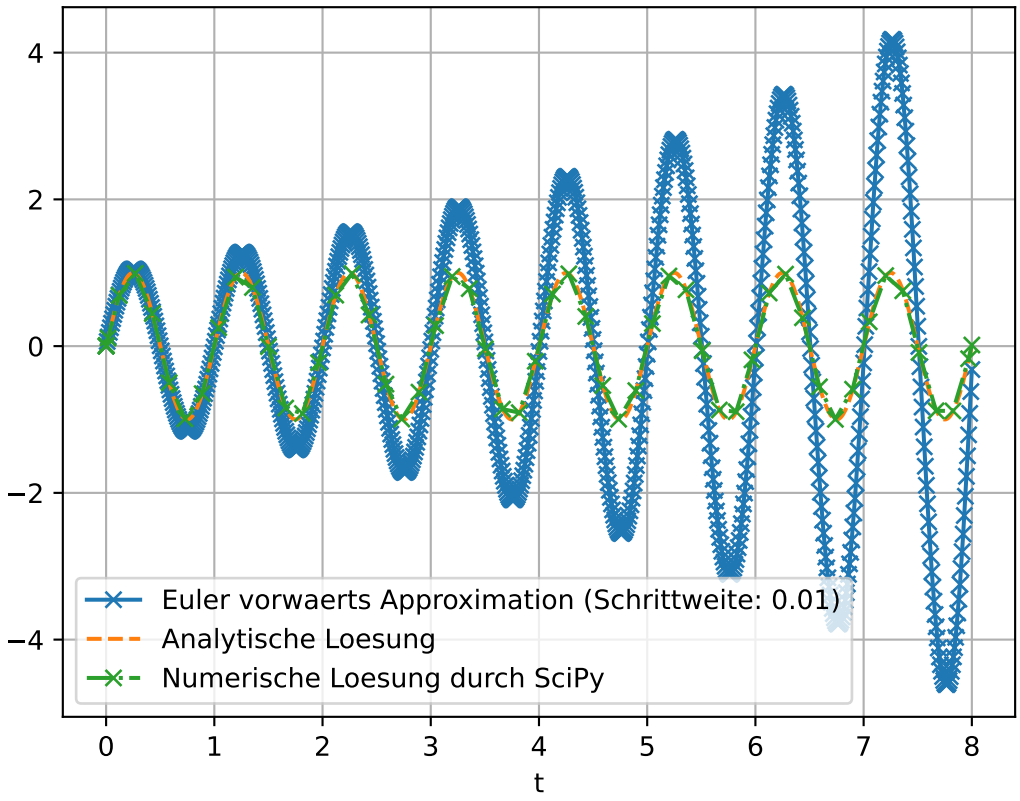
[9]:
```
# Euler vorwaerts fuer die zweite Differentialgleichung
# Hier benoetigen wir die Zustandsraumdarstellung

# Parameter omega und Systemmatrix definieren
omega = 2.0*pi
A = array([[0.0, 1.0], [-omega**2, 0.0]])

# Rechte Seite der Differenzengleichung
def next_x(x_k, step_size):
    B = A*step_size + eye(2)
    x_k_plus_1 = matmul(B, x_k)
    return x_k_plus_1

# Schrittweite festlegen
delta_t = 0.01

# Anfangsbedingung
x_0 = array([0.0, omega])

# Ende der Integration (in s)
t_f = 8.0

x = []
t = []

# Numerische Integration durchfuehren
```

```
current_t = 0.0
current_x = x_0
while current_t <= t_f:
    t.append(current_t)
    x.append(current_x)
    current_x = next_x(current_x, delta_t)
    current_t += delta_t
```

[10]:
```
# Analytische Loesung
t_analytic = linspace(0.0, t_f, 1000)
x_analytic = [sin(omega*tau) for tau in t_analytic]
```

[11]:
```
# Numerische Integration mit SciPy

# Definition der rechten Seite (right hand side, rhs) der
# Differentialgleichung (ordinary differential equation, ODE)
# dx/dt = f(t, x)
def ode_rhs(t, x):
    x_1 = x[0]
    x_2 = x[1]
    x_1_dot = x_2
    x_2_dot = -omega**2 * x_1
    x_dot = array([x_1_dot, x_2_dot])
    return x_dot

scipy_integration_result = solve_ivp(ode_rhs, [0, t_f], x_0)
```

[12]:
```
# Ergebnisse darstellen
plt.figure(1)
plt.clf()
plt.plot(t, [xi[0] for xi in x], '-x', label=f'Euler vorwaerts Approximation␣
 ↪(Schrittweite: {delta_t})')
plt.plot(t_analytic, x_analytic, '--', label='Analytische Loesung')
plt.plot(scipy_integration_result.t, scipy_integration_result.y[0,:], '-.x',␣
 ↪label='Numerische Loesung durch SciPy')
plt.grid(True)
plt.legend()
plt.xlabel('t')
plt.show()
```


4
2
0
−2
−4
0
1
2
3
4
5
6
7
8
t
Euler vorwaerts Approximation (Schrittweite: 0.01)
Analytische Loesung
Numerische Loesung durch SciPy